Project N^o: **FP6-015905**

Project Acronym: **MOBIUS**

Project Title: **Mobility, Ubiquity and Security**

Instrument: **Integrated Project**

Priority 2: **Information Society Technologies**

Future and Emerging Technologies

Deliverable D3.6

Intermediate report on modular verification

Due date of deliverable: 2008-08-31 (T0+36)

Actual submission date: 2008-10-15

Start date of the project: **1 September 2005** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **ETH**

Project co-funded by the European Commission in the Sixth Framework Programme (2002-2006)		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Contributions

Site	Contributed to Chapter
ETH	1, 2, 3, 4
IC	1, 2, 4

This document was written by Ádám Darvas (ETH), Sophia Drossopoulou (IC), Adrian Francalanza (formerly IC), Peter Müller (ETH), Arsenii Rudich (ETH), and Alexander Summers (IC).

Executive Summary:
Intermediate report on modular verification

This document summarises deliverable D3.6 of project FP6-015905 (MOBIUS), co-funded by the European Commission within the Sixth Framework Programme. Full information on this project, including this deliverable, is available online at `http://mobius.inria.fr`.

Following the goals of Task 3.4 to develop a practical verification technique for object invariants that can handle interesting implementation patterns including inheritance, call-backs, recursive object structures, and concurrency, the Mobius consortium has made significant contributions to the area of modular verification. These contributions appear in diverse publication venues such as: European Conference on Object-Oriented Programming "ECOOP" 2008, Workshop on Formal Techniques for Java-like Programs "FTfJP" 2008, International Symposium on Formal Methods "FM" 2008, and International Joint Conference on Automatic Reasoning "IJCAR" 2008. This deliverable contains an overview of the results and serves as a chart for Mobius' contributions to this area, including pointers to specific contributions for further details.

The following results are in the main focus of the deliverable:

- Several visible-state verification techniques for object invariants have been proposed. It is difficult to compare these techniques and ascertain their soundness because of differences in restrictions on programs and invariants, in the use of advanced type systems (e.g., ownership types), in the meaning of invariants, and in proof obligations.

  We have developed a unified framework for such techniques [19]. We distilled seven parameters that characterise a verification technique, and identify sufficient conditions on these parameters which guarantee soundness. We instantiated our framework with three verification techniques from the literature, and use it to assess soundness and compare expressiveness.

- We used the unified framework from the previous item to formalise a generalisation of the Visibility Technique [31] to cater for static fields and methods [37].

  In order to cater for mutable static fields, we extend this topology to multiple trees (a forest), where each tree is rooted in a class. This allows classes to naturally own object instances as their static fields. We described how to extend the Visibility Technique to this topology, incorporating extra flexibility for the treatment of static methods.

- We have developed a technique to check well-formedness of contracts [34]. We give proof obligations that are sufficient to guarantee the existence of a model for the specification of pure methods. We improve over earlier work by providing a systematic solution including a soundness result and by supporting more forms of recursive specifications.

- We have developed a procedure which generates well-definedness conditions that grow linearly with respect to the input formula [15]. We also present empirical results that demonstrate the improvements made.

Contents

Chapter 1

Introduction

This deliverable reports on intermediate progress on modular verification (Task 3.4). The work carried out in Task 3.4 is reported in four publications [19, 37, 34, 15], and includes contributions from the MOBIUS partners involved in Task 3.4, namely ETH and IC[1]. Here we present an overview of the achievements, and we refer to the publications for the full details.

Modularity in verification is crucial if verification of realistic applications is to be feasible in practice. Program correctness and security rely on, among other things, object invariants [25]. For instance, a dynamically loaded class should not bring the system in an inconsistent state. Whereas modular verification of simple pre-post specifications for methods is well understood [18, 28], reasoning about object invariants in the presence of aliasing, subclassing and call-backs is still an active research area.

Several visible-state verification techniques for object invariants have been proposed. It is difficult to compare these techniques and ascertain their soundness because of differences in restrictions on programs and invariants, in the use of advanced type systems (e.g., ownership types), in the meaning of invariants, and in proof obligations.

We develop a unified framework for such techniques. Chapter 2 presents an overview of the unified framework for visible-state verification techniques for object invariants [19] and its application to formalisation of an extension of the Visibility Technique to cater for static fields and methods [37].

One of the main goals of the MOBIUS project is to enable the Proof Carrying Code (PCC) paradigm. A specification's well-formedness is crucial for the PCC paradigm. Ill-formedness of the specification can result in inconsistency of underlying axioms which leads to unsoundness of the PCC. The proposed technique can be used to prevent PCC unsoundness due to specification ill-formedness.

Chapter 3 considers a technique to check well-formedness of contracts. Contracts are well-formed if they respect the partiality of operations, and they enable a consistent encoding of pure methods in a program logic [34]. A new efficient technique for dealing with partiality is presented [15].

Appendix A collates the four publications that constitute this deliverable.

[1]RUN withdraws from Task 3.4, moving work and man months from task 3.4 to task 3.3.

Chapter 2

Object invariants

Object invariants play a crucial role in the verification of object-oriented programs, and have been an integral part of all major contract languages such as Eiffel [30], the Java Modeling Language JML [24], and Spec# [4]. Object invariants express consistency criteria for objects, ranging from simple properties of single objects (for instance, that a field is non-null) to complex properties of whole object structures (for instance, the sorting of a tree).

While the basic idea of object invariants is simple, verification techniques for practical OO-programs face challenges. These challenges are made more daunting by the common expectation that classes should be verified without knowledge of their clients and subclasses. The three main challanges are:

Call-backs: Methods that are called while the invariant of an object o is temporarily broken might call back into o and find o in an inconsistent state.

Multi-object invariants: When the invariant of an object p depends on the state of another object o, modifications of o potentially break the invariant of p. In particular, when verifying o, the invariant of p may not be known and, if not, cannot be expected to be preserved.

Subclassing: When the invariant of a subclass D refers to fields declared in a superclass C then methods of C can break D's invariant by assigning to these fields. In particular, when verifying a class, its subclass invariants are not known in general, and so cannot be expected to be preserved.

Several verification techniques address some or all of these challenges [3, 6, 21, 23, 26, 29, 31, 33]. They share many commonalities, but differ in the following important aspects:

1. *Invariant semantics:* Which invariants are expected to hold when?
2. *Invariant restrictions:* Which objects may invariants depend on?
3. *Proof obligations:* What proofs are required, and where?
4. *Program restrictions:* Which objects' methods/fields may be called/updated?
5. *Type systems:* What syntactic information is used for reasoning?
6. *Specification languages:* What syntax is used to express invariants?
7. *Verification logics:* How are invariants proved?

These differences, together with the fact that most verification techniques are not formally specified, complicate the comparison of verification techniques, and hinder the understanding of why these techniques satisfy claimed properties such as soundness. For these reasons, it is hard to decide which technique to adopt, or to develop new sound techniques.

We developed a unified framework for verification techniques for object invariants [19]. This framework formalises verification techniques in terms of seven parameters, which abstract away from differences pertaining to language features (type system, specification language, and logics) and highlight characteristics intrinsic to the techniques, thereby aiding comparisons. Subsets of these parameters describe aspects applicable to all verification techniques; for example, a generic definition of *soundness* is given in terms of two framework parameters, expressivity is captured by three other parameters.

We used the unified framework to formalise an extension of the Visibility Technique (VT for short) [31]to cater for static fields and methods [37].

When adding statics to verification, one needs to address the following questions:

1. Where in the topology do static fields appear?
2. May instance methods update static fields?
3. May static invariants mention the fields of objects of their class?
4. May instance invariants mention static fields of their class, or of other classes?
5. Can static methods break invariants of objects, and if so, of which objects?
6. Can instance methods break static invariants, and if so, of which classes?
7. What proof obligations are necessary before a call to a static method?
8. What proof obligations are necessary before a call to an instance method?

We explored these questions in the context of VT, and extended the technique and heap topology to handle static fields. In the process, we encountered a potential source of callbacks not present in VT, and solved this problem. We developed an approach involving a combination of effect annotations and refinements to the heap topology using levels. We extended the technique to allow more expressive invariants.

The fundamental premise of this technique is that classes should be able to own objects in the same way that other objects can. For example, if the behaviour of a class depends on a static field (to manage object creation, etc.) then this static field naturally 'belongs' to the inner workings of the class: its representation. This gives a natural interpretation of static rep fields: they should be treated analogously to instance rep fields, but with a class as their owner [27].

Thus, we extended our heap topology to include classes. Classes are the 'roots' of trees in our topology. As there are generally several classes in a program, our topology should allow for several such trees; we work with a *forest*. Furthermore, with classes acting as roots, there is no longer a need for an abstract root entity; these class-rooted trees make up the entire picture. Note that there are no objects at the 'same level' as the class entities, and classes do not have owners. In this paper, we do not consider a notion of static peer fields.

We interpret static fields and methods as instance fields and methods of the corresponding class object. That is, the class object (or class for short) is the receiver for an execution of a static method. We expect that modifications to static fields will be achieved by calling a static method of the class that declares the field. In other words, static methods may update the fields of their receiver class, just like instance methods in VT may update fields of their receiver object.

To summarise the ideas:

1. Each point in our heap topology corresponds to either an object or a class.
2. Objects (but not classes) each have exactly one owner (a class or an object).
3. The current receiver (on the stack) can be either an object or a class.

Chapter 3

Checking well-formedness of pure-method specifications

Contract languages such as the Java Modeling Language (JML) [22] and Spec# [5] specify invariants and pre- and postconditions using side-effect free expressions of the programming language. While contract languages are natural for programmers, they pose various challenges when contracts are encoded in the logic of a program verifier or theorem prover, especially when contracts use pure (side-effect free) methods [16]. This chapter addresses two challenges related to pure-method specifications [34].

The first challenge is how to ensure that a specification is *well-defined*, that is, that all partial operations are applied within their domain. For instance method calls are well-defined only for non-null receivers and when the precondition of the method is satisfied. This challenge can be solved by encoding partial functions as under-specified total functions [20]. However, it has been argued that such an encoding is counter-intuitive for programmers, is not well-suited for runtime assertion checking, and assigns meaning to bogus contracts instead of having them rejected by a verifier [10]. Another solution is the use of 3-valued logic, such as LPF [7]. However, 3-valued logic is typically not supported by the theorem provers that are used in program verifiers. We present a technique based on 2-valued logic to check whether a specification satisfies all partiality constraints. If the check fails, the specification is rejected.

The second challenge is how to ensure that a specification is consistent. In order to reason about contracts that contain pure-method calls, pure methods must be encoded in the logic of the program verifier. This is typically done by introducing an uninterpreted function symbol for each pure method m, whose properties are axiomatised based on m's contract and object invariants [12, 16]. A specification is *consistent* if this axiomatisation is free from contradictions. Consistency is crucial for soundness. We present a technique to check consistency by showing that the contracts of pure methods are satisfiable and well-founded if they are recursive. If the consistency check fails, the specification is rejected.

An inconsistent specification of a method m is not necessarily detected during the verification of m's implementation [16]: (1) m might be abstract; (2) partial correctness logics allow one to verify m w.r.t. an unsatisfiable specification if m's implementation does not terminate; (3) any implementation could be trivially verified based on inconsistent axioms stemming from inconsistent pure-method specifications; this is especially true for recursion, when the axiom for m is needed to verify its implementation. These reasons justify the need for verifying consistency of specifications independently of implementations.

We show well-formedness of specifications by posing proof obligations to ensure: (1) that partial operations are applied within their domains, (2) the existence of a possible result value for each pure method, and (3) that recursive specifications are well-founded. In order to deal with dependencies between pure methods, we determine a dependency graph, which we process bottom-up. Thereby, one can use the properties of a method m to prove the proof obligations for the methods using m.

To deal with partiality, we interpret specifications in 3-valued logic. However, we want to support standard theorem provers, which typically use 2-valued logic and total functions [32, 17]. Therefore, we express the proof obligations in 2-valued logic by applying a well-definedness condition generator to the specification expressions. *Well-definedness conditions* validity ensures that all formulas at hand can be evaluated to either **true** or **false**.

The literature [8, 35, 1, 9] proposes the procedure $\mathcal{D}$ to generate well-definedness conditions. The procedure is *complete* [8, 9], that is, the well-definedness condition generated from a formula is provable if and only if the formula is well-defined.

Due to the exponential blow-up of well-definedness conditions, the $\mathcal{D}$ procedure is not used in practice [8, 35, 1]. Instead, another procedure $\mathcal{L}$ is used, which generates much smaller conditions with linear growth, but which is *incomplete*. That is, the procedure may generate unprovable well-definedness conditions for well-defined formulas.

We developed a new procedure $\mathcal{Y}$ [15], which unifies the advantages of $\mathcal{D}$ and $\mathcal{L}$, while eliminating their weaknesses. That is, (1) $\mathcal{Y}$ yields well-definedness conditions that *grow linearly* with respect to the size of the input formula, and (2) $\mathcal{Y}$ is equivalent to $\mathcal{D}$, and therefore complete and insensitive to the order of sub-formulas. To our knowledge, this is the first procedure that has *both* of these two properties.

The definition of the new procedure is very intuitive and straightforward. We prove that it is equivalent with $\mathcal{D}$ in two ways: (1) in a syntactical manner, as $\mathcal{D}$ was derived in [1], and (2) in a semantical way, as $\mathcal{D}$ was introduced in [8].

We proved the following soundness result: If all proof obligations for the pure methods of a program are proved then there is a partial model for the axiomatisation of these pure methods. In other words, we guarantee that the partiality constraints are satisfied and the axiomatisation is consistent.

Our approach differs from existing solutions for theorem provers [13, 32], where consistency is typically enforced by restricting specifications to conservative extensions, but no checks are performed for axioms. Since specifications of pure methods are axiomatic, the approach of conservative extensions is not applicable to contract languages. Moreover, theorem provers require the user to resolve dependencies by ordering the elements of a theory appropriately. We determine this order automatically using a dependency graph.

Our approach improves on existing solutions for program verifiers in three ways. First, it supports (mutually) recursive specifications, whereas in previous work recursive specifications are severely restricted [16, 14]. Second, our approach allows us to use the specification of one method to prove well-formedness of another, which is needed in many practical examples. Such dependencies are not discussed in previous work [11, 16] and are not supported by program verifiers that perform consistency checks, such as Spec#. Neglecting dependencies leads to the rejection of well-formed specifications. Third, we prove consistency for the axiomatisation of pure methods; such a proof is either missing in earlier work [11, 14] or only presented for a very restricted setting [16].

Chapter 4

Conclusions

This chapter summarises the results and briefly discusses their impact on modular verification.

Results There is rich evidence that we have achieved the task goals :

- We presented a unified framework that describes verification techniques for object invariants in terms of seven parameters and separates verification concerns from those of the underlying type system. We identified sufficient conditions on the framework parameters that guarantee soundness, and we proved a universal soundness theorem. The result have been presented in European Conference on Object-Oriented Programming "ECOOP" 2008 [19].
- We have outlined a verification technique based on VT, catering for static fields, methods, and invariants. In the process, we extended the usual heap topology of ownership types, and tackled potential callbacks through a combination of effects, levels, and the owner-as-modifier discipline. The result have been presented in Workshop on Formal Techniques for Java-like Programs "FTfJP" 2008 [37].
- We presented a new technique to check the well-formedness of specifications. We showed how to incrementally construct a model for the specification, which guarantees that the partiality constraints of operations are respected and that the axiomatisation of pure methods is consistent. contract language, logic, or backend theorem prover. The result have been presented in International Symposium on Formal Methods "FM" 2008 [34].
- We proposed a new procedure $\mathcal{Y}$ to generate well-definedenss conditions. $\mathcal{Y}$ is complete and yields formulas that grow linearly with respect to the size of the input formula. Our procedure has been used to enforce well-formedness of invariants and method specifications. The result have been presented in International Joint Conference on Automatic Reasoning "IJCAR" 2008 [15].

Further development within the project Further development within the project will be reported on in the final report on modular verification, deliverable D3.9.

- ETH and IC will develop a verification technique for non-hierarchic object structures. They will combine their work on hierarchic (ownership-based) structures with recent results on dynamic frames [36] and regions [2]. The aim is a verification technique that provides the simplicity of ownership-based verification with the flexibility of dynamic frames and regions. A comparison of the four major existing verification techniques for object structures (dynamic frames, ownership, regions, separation logic) will be an intermediate result.
- ETH will continue their work on the well-definedness of specifications by extending it to model classes, which encode mathematical structures. Program verifiers map model classes to their underlying logics. Flaws in a model class or in the mapping can easily lead to unsoundness and incompleteness. ETH will continue to develop a technique for finding such flaws.

Acknowledgments We are grateful to Erik Poll and Joseph Kiniry for helpful discussions and for their insightful comments.

Bibliography

[1] J. Abrial and L. Mussat. On using conditional definitions in formal theories. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002*, volume 2272 of *Lecture Notes in Computer Science*, pages 242–269. Springer-Verlag, 2002.

[2] A. Banerjee, D. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In J. Vitek, editor, *European Conference on Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 387–411. Springer-Verlag, 2008.

[3] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 151–171. Springer-Verlag, 2005.

[5] M. Barnett, R. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.

[6] M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared state. In D. Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 54–84. Springer-Verlag, 2004.

[7] H. Barringer, J. H. Cheng, and C. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1998.

[8] P. Behm, L. Burdy, and J. Meynadier. Well Defined B. In D. Bert, editor, *International B Conference*, volume 1393 of *LNCS*, pages 29–45. Springer-Verlag, 1998.

[9] S. Berezin, C. Barrett, I. Shikanian, M. Chechik, A. Gurfinkel, and D.Dill. A practical approach to partial functions in CVC Lite. In *Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2004.

[10] P. Chalin. Are the logical foundations of verifying compiler prototypes matching user expectations? *Formal Aspects of Computing*, 19(2):139–158, 2007.

[11] P. Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *ICSE*, pages 23–33. IEEE Computer Society, 2007.

[12] D. R. Cok. Reasoning with specifications containing method calls in JML. *Journal of Object Technology*, 4(8):77–103, 2005.

[13] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. *A Tutorial Introduction to PVS*, April 1995.

[14] Á. Darvas and R. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, volume 4422 of *LNCS*, pages 336–351. Springer-Verlag, 2007.

[15] Á. Darvas, F. Mehta, and A. Rudich. Efficient well-definedness checking. In *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2008.

[16] Á. Darvas and P. Müller. Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology (JOT)*, 5(5):59–85, June 2006.

[17] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the Association of Computing Machinery*, 52(3):365–473, 2005.

[18] K. K. Dhara. Behavioral subtyping in object-oriented languages. Technical Report 97-09, Iowa State University, May 1997.

[19] S. Drossopoulou, A. Francalanza, Peter Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *European Conference of Object Oriented Programming*, 2008.

[20] D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 366–373. Springer-Verlag, 1995.

[21] K. Huizing and R. Kuiper. Verification of object-oriented programs using class invariants. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 1783 of *Lecture Notes in Computer Science*, pages 208–221. Springer-Verlag, 2000.

[22] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[23] G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *International Conference on Software Engineering (ICSE)*, pages 385–395. IEEE, 2007.

[24] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, February 2007. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.

[25] K. R. M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995.

[26] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004. Available from `www.sct.inf.ethz.ch/publications/index.html`.

[27] R. Leino and P. Müller. Modular verification of static class invariants. In *Formal Methods*, 2005.

[28] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[29] Y. Lu, J. Potter, and J. Xue. Object Invariants and Effects. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 202–226. Springer-Verlag, 2007.

[30] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

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

[32] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic.* Springer-Verlag, 2002.

[33] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, 1997.

[34] A. Rudich, Á. Darvas, and P. Müller. Checking well-formedness of pure-method specifications. In J. Cuellar and T. Maibaum, editors, *Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 68–83. Springer-Verlag, 2008.

[35] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[36] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for Java-like programs based on dynamic frames. In J. L. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer-Verlag, 2008.

[37] Alexander J. Summers, S. Drossopoulou, and P. Müller. A universe-type-based verification technique for mutable static fields and methods (work in progress). In *Workshop on Formal Techniques for Java Programs*, 2008.

Appendix A

Copies of Publications

A Unified Framework for Verification Techniques for Object Invariants

S. Drossopoulou(1), A. Francalanza(2), P. Müller(3), and A. J. Summers(1)

(1) Imperial College London,
(2) University of Southampton,
(3) Microsoft Research, Redmond

Abstract. Object invariants define the consistency of objects. They have subtle semantics because of call-backs, multi-object invariants and subclassing. Several visible-state verification techniques for object invariants have been proposed. It is difficult to compare these techniques and ascertain their soundness because of differences in restrictions on programs and invariants, in the use of advanced type systems (*e.g.*, ownership types), in the meaning of invariants, and in proof obligations.
We develop a unified framework for such techniques. We distil seven parameters that characterise a verification technique, and identify sufficient conditions on these parameters which guarantee soundness. We instantiate our framework with three verification techniques from the literature, and use it to assess soundness and compare expressiveness.

1 Introduction

Object invariants play a crucial role in the verification of object-oriented programs, and have been an integral part of all major contract languages such as Eiffel [25], the Java Modeling Language JML [17], and Spec# [2]. Object invariants express consistency criteria for objects, ranging from simple properties of single objects (for instance, that a field is non-null) to complex properties of whole object structures (for instance, the sorting of a tree).

While the basic idea of object invariants is simple, verification techniques for practical OO-programs face challenges. These challenges are made more daunting by the common expectation that classes should be verified without knowledge of their clients and subclasses:

Call-backs: Methods that are called while the invariant of an object o is temporarily broken might call back into o and find o in an inconsistent state.

Multi-object invariants: When the invariant of an object p depends on the state of another object o, modifications of o potentially break the invariant of p. In particular, when verifying o, the invariant of p may not be known and, if not, cannot be expected to be preserved.

Subclassing: When the invariant of a subclass D refers to fields declared in a superclass C then methods of C can break D's invariant by assigning to these fields. In particular, when verifying a class, its subclass invariants are not known in general, and so cannot be expected to be preserved.

Several verification techniques address some or all of these challenges [1, 3, 14, 16, 18, 23, 26, 27, 31]. They share many commonalities, but differ in the following important aspects:

1. *Invariant semantics:* Which invariants are expected to hold when?
2. *Invariant restrictions:* Which objects may invariants depend on?
3. *Proof obligations:* What proofs are required, and where?
4. *Program restrictions:* Which objects' methods/fields may be called/updated?
5. *Type systems:* What syntactic information is used for reasoning?
6. *Specification languages:* What syntax is used to express invariants?
7. *Verification logics:* How are invariants proved?

These differences, together with the fact that most verification techniques are not formally specified, complicate the comparison of verification techniques, and hinder the understanding of why these techniques satisfy claimed properties such as soundness. For these reasons, it is hard to decide which technique to adopt, or to develop new sound techniques.

In this paper, we present a unified framework for verification techniques for object invariants. This framework formalises verification techniques in terms of seven parameters, which abstract away from differences pertaining to language features (type system, specification language, and logics) and highlight characteristics intrinsic to the techniques, thereby aiding comparisons. Subsets of these parameters describe aspects applicable to all verification techniques; for example, a generic definition of *soundness* is given in terms of two framework parameters, expressivity is captured by three other parameters.

We concentrate on techniques that require invariants to hold in the pre-state and post-state of a method execution (often referred to as *visible states* [27]) while temporary violations between visible states are permitted. These techniques constitute the vast majority of those described in the literature.

Contributions. The contributions of this paper are:

1. We present a unified formalism for object invariant verification techniques.
2. We identify conditions on the framework that guarantee soundness of a verification technique.
3. We separate type system concerns from verification strategy concerns.
4. We show how our framework describes some advanced verification techniques for visible state invariants.
5. We prove soundness for a number of techniques, and, guided by our framework, discover an unsoundness in one technique.

Our framework allows the extraction of comparable data from techniques that were presented using different concepts, terminology and styles. Comparative value judgements concerning the techniques are beyond the scope of our paper.

Outline. Sec. 2 gives an overview of our work, explaining the important concepts. Sec. 3 formalises program and invariant semantics. Sec. 4 describes our framework and defines soundness. Sec. 5 instantiates our framework with existing verification techniques. Sec. 6 presents sufficient conditions for a verification technique to be sound, and states a general soundness theorem. Sec. 7 discusses related work. Proofs and more details are in the companion report [8]. This paper follows our FOOL paper [7], but provides more explanations and examples.

2 Example and Approach

Example. Consider a scenario, in which a Person holds an Account, and has a salary. An Account has a balance, an interestRate and an associated DebitCard. This example will be used throughout the paper. We give the code in Fig. 1.

```
class Account {
 Person holder;
 DebitCard card;
 int balance, interestRate;

 // invariant I1: balance < 0 ==>
       interestRate == 0;
 // invariant I2: card.acc == this;

 void withdraw(int amount) {
   balance -= amount;
   if (balance < 0) {
     interestRate = 0;
     this.sendReport();
   }
 }

 void sendReport()
   { holder.notify(); }
}

class SavingsAccount
            extends Account {
 // invariant I3: balance >= 0;
}
```

```
class Person {
  Account account;
  int salary;

  // invariant I4:
  //   account.balance + salary > 0;

  void spend(int amount)
    { account.withdraw(amount); }

  void notify()
    { ... }
}

class DebitCard {
  Account acc;
  int dailyCharges;

  // invariant I5:
  //   dailyCharges <= acc.balance;
}
```

Fig. 1. An account example illustrating the main challenges for the verification of object invariants. We assume that fields hold non-null values.

Account's interestRate is required to be zero when the balance is negative (I1). A further invariant (the two can be read as conjuncts of the full invariant for the class) ensures that the DebitCard associated with an account has a consistent reference back to the account (I2). A SavingsAccount is a special kind of Account, whose balance must be positive (I3). Person's invariant (I4) requires that the sum of salary and account's balance is positive. Finally, DebitCard's invariant (I5) requires dailyCharges not to exceed the balance of the associated account. Thus, I2, I4, and I5 are multi-object invariants.

To illustrate the challenges faced by verification techniques, suppose that p is an object of class Person, which holds the Account a with DebitCard d:

Call-backs: When p executes its method spend, this results in a call of withdraw on a, which (via a call to sendReport) eventually calls back notify on p; the call notify might reach p in a state where I4 does not hold.

Multi-object invariants: When a executes its method withdraw, it may temporarily break its invariant I1, since its balance is debited before any corresponding change is made to its interestRate. This violation is not important according to the visible state semantics; the **if** statement immediately afterwards ensures that the invariant is restored before the next visible state. However, by making an unrestricted reduction of the account balance, the method potentially breaks the invariants of other objects as well. In particular, p's invariant I4, and d's invariant I5 may be broken.

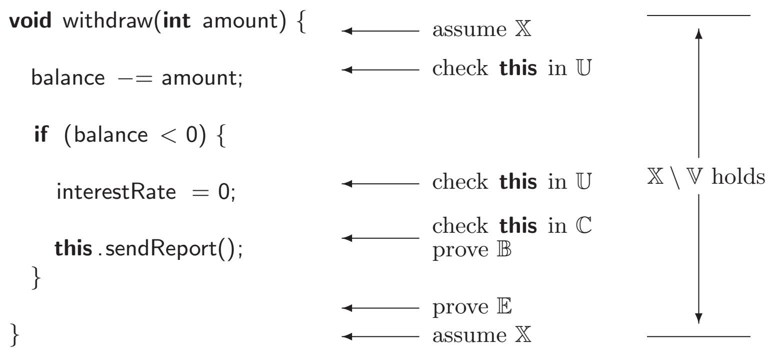
Subclassing: Further to the previous point, if a is a SavingsAccount, then calling the method withdraw may break the invariant I3, which was not necessarily known during the verification of class Account.

These points are addressed in the literature by striking various trade-offs between the differing aspects listed in the introduction.

Approach. Our framework uses seven parameters to capture the first four aspects in which verification techniques differ, *i.e.,* invariant semantics, invariant restrictions, proof obligations and program restrictions. To describe these parameters we use two abstract notions, which we call *regions* and *properties.* A *region* (when interpreted semantically) describes a set of objects (*e.g.*, those on which a method may be called), while a property describes a set of invariants (*e.g.*, the invariants that have to be proven before a method call). We deal with the aspects identified in the previous section as follows:

1. *Invariant semantics:* The property $\mathbb{X}$ describes the invariants e$\underline{\text{x}}$pected to hold in visible states. The property $\mathbb{V}$ describes the invariants *$\underline{v}$ulnerable* to a given method, *i.e.,* those which may be broken while the method executes.
2. *Invariant restrictions:* The property $\mathbb{D}$ describes the invariants that may $\underline{\text{d}}$epend on a given heap location. This also characterises indirectly the locations an invariant may depend on.
3. *Proof obligations:* The properties $\mathbb{B}$ and $\mathbb{E}$ describe the invariants that must be proven to hold $\underline{\text{b}}$efore a method call and at the $\underline{\text{e}}$nd of a method body, respectively.
4. *Program restrictions:* The regions $\mathbb{U}$ and $\mathbb{C}$ describe the permitted receivers for field $\underline{\text{u}}$pdates and method $\underline{\text{c}}$alls, respectively.
5. *Type systems:* We parameterise our framework by the type system. We state requirements on the type system, but leave abstract its concrete definition. We require that types are formed of a region-class pair so that we can handle types that express heap topologies (such as ownership types).
6. *Specification languages:* Rather than describing invariants concretely, we assume a judgement that expresses that an object satisfies the invariant of a class in a heap.
7. *Verification logics:* We express proof obligations via a special construct prv $\mathbb{p}$, which throws an exception if the invariants in property $\mathbb{p}$ cannot be proven, and has an empty effect otherwise. We leave abstract how the actual proofs are constructed and checked.

Fig. 2 illustrates the parameters of our framework by annotating the body of the method withdraw. $\mathbb{X}$ may be assumed to hold in the pre- and post-states of the method. Between these visible states, some object invariants may be broken (so long as they fall within $\mathbb{V}$), but $\mathbb{X}\setminus\mathbb{V}$ is known to hold throughout the method body. Field updates and method calls are allowed if the receiver object (here, **this**) is in $\mathbb{U}$ and $\mathbb{C}$, respectively. Before a method call, $\mathbb{B}$ must be proven. At the end of the method body, $\mathbb{E}$ must be proven. Finally, $\mathbb{D}$ (not shown in Fig. 2) constrains the effects of field updates on invariants. Thus, assignments to balance and interestRate affect at most $\mathbb{D}$.



```
void withdraw(int amount) {

  balance −= amount;

  if (balance < 0) {

    interestRate = 0;

    this.sendReport();
  }

}
```

assume $\mathbb{X}$
check **this** in $\mathbb{U}$
check **this** in $\mathbb{U}$
check **this** in $\mathbb{C}$
prove $\mathbb{B}$
prove $\mathbb{E}$
assume $\mathbb{X}$
$\mathbb{X} \setminus \mathbb{V}$ holds

Fig. 2. Role of framework parameters for method withdraw from Fig. 1.

The number of parameters reflects the variety of concepts used by verification techniques, such as accessibility of fields, purity, helper methods, ownership, and effect specifications. For instance, $\mathbb{V}$ would be redundant if all methods were to re-establish the invariants they break; in such a setting, a method could break invariants only through field updates, and $\mathbb{V}$ could be derived from $\mathbb{U}$ and $\mathbb{D}$. However, in general, methods may break but not re-establish invariants.

The seven parameters capture concepts explicitly or implicitly found in all verification techniques, defined either through words [27, 14, 16, 31] or typing rules [23]. For example, $\mathbb{V}$ is implicit in [27], but is crucial for their soundness argument. $\mathbb{X}$ and $\mathbb{V}$ are explicit in [23], while $\mathbb{U}$ and $\mathbb{C}$ are implicitly expressed as constraints in their typing rules. Subsets of these seven parameters characterise verification technique concepts e.g., soundness (through $\mathbb{X}$ and $\mathbb{V}$), expressiveness ($\mathbb{D}$, $\mathbb{X}$ and $\mathbb{V}$), proof obligations ($\mathbb{B}$ and $\mathbb{E}$).

3 Invariant Semantics

We formalise invariant semantics through an operational semantics, defining at which execution points invariants are required to hold. In order to cater for the different techniques, the semantics is parameterised by properties to express proof obligations and which invariants are expected to hold. In this section, we focus on the main ideas of our semantics and relegate the less interesting definitions to App. A. We assume sets of identifiers for class names Cls, field

$$
\begin{array}{rlllllll}
e ::= & \mathsf{this} & \textit{(this)} & \mid x & \textit{(variable)} & \mid \mathsf{null} & \textit{(null)} \\
\mid & \mathsf{new}\, t & \textit{(new object)} & \mid e.f & \textit{(access)} & \mid e.f = e & \textit{(assignment)} \\
\mid & e.m(e) & \textit{(method call)} & \mid e\, \mathsf{prv}\, \mathbb{p} & \textit{(proof annotat.)} & & \\
 & & & & & & \\
e_r ::= & \ldots & \textit{(as source exprs.)} & \mid v & \textit{(value)} & \mid \mathsf{verfExc} & \textit{(verif exc.)} \\
\mid & \mathsf{fatalExc} & \textit{(fatal exc.)} & \mid \sigma \cdot e_r & \textit{(nested call)} & \mid \mathsf{call}\, e_r & \textit{(launch)} \\
\mid & \mathsf{ret}\, e_r & \textit{(return)} & & & &
\end{array}
$$

Fig. 3. Source and runtime expression syntax.

names F_{LD}, and method names M_{THD}, and use variables $c \in \text{C}_{\text{LS}}$, $f \in \text{F}_{\text{LD}}$ and $m \in \text{M}_{\text{THD}}$.

Runtime Structures. A *runtime structure* is a tuple consisting of a set of heaps H_{P}, a set of addresses A_{DR}, and a set of values $\text{V}_{\text{AL}} = \text{A}_{\text{DR}} \cup \{\mathsf{null}\}$, using variables $h \in \text{H}_{\text{P}}$, $\iota \in \text{A}_{\text{DR}}$, and $v \in \text{V}_{\text{AL}}$. A runtime structure provides the following operations. The operation $\mathit{dom}(h)$ represents the domain of the heap. $\mathit{cls}(h, \iota)$ yields the class of the object at address ι. The operation $\mathit{fld}(h, \iota, f)$ yields the value of a field f of the object at address ι. Finally, $\mathit{upd}(h, \iota, f, v)$ yields the new heap after a field update, and $\mathit{new}(h, \iota, t)$ yields the heap and address resulting from the creation of a new object of type t. We leave abstract how these operations work, but require properties about their behaviour, for instance that *upd* only modifies the corresponding field of the object at the given address, and leaves the remaining heap unmodified. See Def. 9 in App. A for details.

A stack frame $\sigma \in \text{S}_{\text{TK}} = \text{A}_{\text{DR}} \times \text{A}_{\text{DR}} \times \text{M}_{\text{THD}} \times \text{C}_{\text{LS}}$ is a tuple of a receiver address, an argument address, a method identifier, and a class. The latter two indicate the method currently being executed and the class where it is defined.

Regions, Properties and Types. A region $\mathbb{r} \in \text{R}$ is a syntactic representation for a set of objects; a property $\mathbb{p} \in \text{P}$ is a syntactic representation for a set of assertions about particular objects. It is crucial that our syntax is parametric with the specific regions and properties; we use different regions and properties to model different verification techniques.[1]

We define a type $t \in \text{T}_{\text{YP}}$, as a pair of a region and a class. The region allows us to cater for types that express the topology of the heap, without being specific about the underlying type system.

Expressions. In Fig. 3, we define source expressions $e \in \text{E}_{\text{XPR}}$. In order to simplify our presentation (but without loss of generality), we restrict methods to always have exactly one argument. Besides the usual basic object-oriented constructs, we include proof annotations $e\,\mathsf{prv}\,\mathbb{p}$. As we will see later, such a proof annotation executes the expression e and then imposes a proof obligation for the invariants characterised by the property $\mathbb{p}$. To maintain generality, we avoid being precise about the actual syntax and checking of proofs.

[1] For example, in Universe types, **rep** and **peer** are regions, while in ownership types, ownership parameters such as X, and also **this**, are regions (more in Sec. 5).

In Fig. 3, we also define runtime expressions $e_r \in \textsc{RExpr}$. A runtime expression is a source expression, a value, a nested call with its stack frame σ, an exception, or a decorated runtime expression. A verification exception $\mathsf{verfExc}$ indicates that a proof obligation failed. A fatal exception $\mathsf{fatalExc}$ indicates that an expected invariant does not hold. Runtime expressions can be decorated with $\mathsf{call}\, e_r$ and $\mathsf{ret}\, e_r$ to mark the beginning and end of a method call, respectively.

In Def. 10 (App. A), we define evaluation contexts, $E[\cdot]$, which describe contexts within one activation record and extend these to runtime contexts, $F[\cdot]$, which also describe nested calls.

Programming Languages. We define a programming language as a tuple consisting of a set $\textsc{Prg}$ of programs, a runtime structure, a set of regions, and a set of properties (see Def. 11 in App. A). Each $\Pi \in \textsc{Prg}$ comes equipped with the following operations. $\mathcal{F}(c, f)$ yields the type of field f in class c as well as the class in which f is declared (c or a superclass of c). $\mathcal{M}(c, m)$ yields the type signature of method m in class c. $\mathcal{B}(c, m)$ yields the expression constituting the body of method m in class c as well as the class in which m is declared. Moreover, there are operators to denote subclasses and subtypes ($<:$), inclusion of regions ($\sqsubseteq$), and interpretation ($[\![\cdot]\!]$) of regions and properties.

The interpretation of a region produces a set of objects. We characterise each invariant by an object-class pair, with the intended meaning that the invariant specified in the class holds for the object.[2] Therefore, the interpretation of a property produces a set of object-class pairs, specifying all the invariants of interest. Regions and properties are interpreted *w.r.t.* a heap, and from the *viewpoint* of a "current object"; therefore, their definitions depend on heap and address parameters: $[\![\ldots]\!]_{h,\iota}$.

Each program also comes with typing judgements $\Gamma \vdash e : t$ and $h \vdash e_r : t$ for source and runtime expressions, respectively. An environment $\Gamma \in \textsc{Env}$ is a tuple of the class containing the current method, the method identifier, and the type of the sole argument.

Finally, the judgement $h \models \iota, c$ expresses that in heap h, the object at address ι satisfies the invariant declared in class c. We define that the judgement trivially holds if the object is not allocated ($\iota \notin \mathit{dom}(h)$) or is not an instance of c ($\mathit{cls}(h, \iota) \not<: c$). We say that the property $\mathbb{p}$ is *valid* in heap h *w.r.t.* address ι if all invariants in $[\![\mathbb{p}]\!]_{h,\iota}$ are satisfied. We denote validity of properties by $h \models \mathbb{p},\, \iota$:

$$h \models \mathbb{p},\, \iota \quad\Leftrightarrow\quad \forall(\iota', c) \in [\![\mathbb{p}]\!]_{h,\iota}.\ h \models \iota', c$$

Operational Semantics. The framework parameter $\mathbb{X}$ describes which invariants are expected to hold at visible states. Given a program Π and a set of properties $\mathbb{X}_{c,m}$, each characterising the property that needs to hold at the beginning and end of a method m of class c, the *runtime semantics* is the relation $\longrightarrow\ \subseteq (\textsc{RExpr} \times \textsc{Hp}) \times (\textsc{RExpr} \times \textsc{Hp})$ defined in Fig. 4.

The first eight rules are standard for object-oriented languages. Note that in rNew, a new object is created using the function *new*, which takes a type as

[2] An object may have different invariants for each of the classes it belongs to [18].

(rThis)
$$\frac{\sigma = (\iota, \_, \_, \_)}{\sigma \cdot \mathsf{this},\ h \longrightarrow \sigma \cdot \iota,\ h}$$

(rVar)
$$\frac{\sigma = (\_, v, \_, \_)}{\sigma \cdot x,\ h \longrightarrow \sigma \cdot v,\ h}$$

(rNew)
$$\frac{\sigma = (\iota, \_, \_, \_) \quad h', \iota' = \mathit{new}(h, \iota, t)}{\sigma \cdot \mathsf{new}\, t,\ h \longrightarrow \sigma \cdot \iota',\ h'}$$

(rDer)
$$\frac{v = \mathit{fld}(h, \iota, f)}{\iota.f,\ h \longrightarrow v,\ h}$$

(rAss)
$$\frac{h' = \mathit{upd}(h, \iota, f, v)}{\iota.f = v,\ h \longrightarrow v,\ h'}$$

(rCxtFrame)
$$\frac{\mathrm{e}_r,\ h \longrightarrow \mathrm{e}'_r,\ h'}{\sigma \cdot \mathrm{e}_r,\ h \longrightarrow \sigma \cdot \mathrm{e}'_r,\ h'}$$

(rCall)
$$\frac{\mathcal{B}(m, \mathit{cls}(h, \iota)) = e, c \quad \sigma = (\iota, v, c, m)}{\iota.m(v),\ h \longrightarrow \sigma \cdot \mathsf{call}\, e,\ h}$$

(rCxtEval)
$$\frac{\sigma \cdot \mathrm{e}_r,\ h \longrightarrow \sigma \cdot \mathrm{e}'_r,\ h'}{\sigma \cdot E[\mathrm{e}_r],\ h \longrightarrow \sigma \cdot E[\mathrm{e}'_r],\ h'}$$

(rLaunch)
$$\frac{\sigma = (\iota, \_, c, m) \quad h \models \mathbb{X}_{c,m},\ \iota}{\sigma \cdot \mathsf{call}\, e,\ h \longrightarrow \sigma \cdot \mathsf{ret}\, e,\ h}$$

(rLaunchExc)
$$\frac{\sigma = (\iota, \_, c, m) \quad h \not\models \mathbb{X}_{c,m},\ \iota}{\sigma \cdot \mathsf{call}\, e,\ h \longrightarrow \sigma \cdot \mathsf{fatalExc},\ h}$$

(rFrame)
$$\frac{\sigma = (\iota, \_, c, m) \quad h \models \mathbb{X}_{c,m},\ \iota}{\sigma \cdot \mathsf{ret}\, v,\ h \longrightarrow v,\ h}$$

(rFrameExc)
$$\frac{\sigma = (\iota, \_, c, m) \quad h \not\models \mathbb{X}_{c,m},\ \iota}{\sigma \cdot \mathsf{ret}\, v,\ h \longrightarrow \mathsf{fatalExc},\ h}$$

(rPrf)
$$\frac{\sigma = (\iota, \_, \_, \_) \quad h \models \mathbb{p},\ \iota}{\sigma \cdot v\ \mathsf{prv}\ \mathbb{p},\ h \longrightarrow \sigma \cdot v,\ h}$$

(rPrfExc)
$$\frac{\sigma = (\iota, \_, \_, \_) \quad h \not\models \mathbb{p},\ \iota}{\sigma \cdot v\ \mathsf{prv}\ \mathbb{p},\ h \longrightarrow \sigma \cdot \mathsf{verfExc},\ h}$$

Fig. 4. Reduction rules of operational semantics.

parameter rather than a class, thereby making the semantics parametric *w.r.t.* the type system: different type systems may use different regions and definitions of *new* to describe heap-topological information. Similarly, *upd* and *fld* do not fix a particular heap representation. Rule rCall describes method calls; it stores the class in which the method body is defined in the new stack frame σ, and introduces the "marker" call e_r at the beginning of the method body.

Our reduction rules abstract away from program verification and describe only its effect. Thus, rLaunch, rLaunchExc, rFrame, and rFrameExc check whether $\mathbb{X}_{c,m}$ is valid at the beginning and end of any execution of a method m defined in class c, and generate a fatal exception, fatalExc, if the check fails. This represents the visible state semantics discussed in the introduction. Proof obligations e prv $\mathbb{p}$ are verified once e reduces to a value (rPrf and rPrfExc); if $\mathbb{p}$ is not found to be valid, a verification exception verfExc is generated.

Verification using visible state semantics amounts to showing all proof obligations in some program logic, based on the assumption that expected invariants hold in visible states. Informally then, a specific verification technique described in our framework is sound if it guarantees that a fatalExc is never encountered. Verification technique soundness does allow verfExc to be generated, but this will never happen in a correctly verified program. We give a formal definition of soundness at the end of the next section.

This semantics allows us to be parametric *w.r.t.* the syntax of invariants and the logic of proofs. We also define properties that permit us to be parametric *w.r.t.* a sound type system (*cf.* Def. 15 in App. A). Thus, we can concentrate entirely on verification concerns.

4 Verification Techniques

A verification technique is essentially a 7-tuple, where the *components* of the tuple provide instantiations for the seven parameters of our framework. These instantiations are expressed in terms of the regions and properties provided by the programming language. To allow the instantiations to refer to the program (for instance, to look up field declarations), we define a verification technique as a mapping from programs to 7-tuples.

Definition 1 *A* verification technique $\mathcal{V}$ for a programming language *is a mapping from programs into a tuple:*

$$\mathcal{V} \;:\; \textsc{Prg} \rightarrow \textsc{eXp} \times \textsc{Vul} \times \textsc{Dep} \times \textsc{Pre} \times \textsc{End} \times \textsc{Upd} \times \textsc{Cll}$$

where

$$\begin{array}{llll}
\mathbb{X} \in & \textsc{eXp} = \textsc{Cls} \times \textsc{Mthd} \rightarrow \mathrm{P} & \mathbb{V} \in & \textsc{Vul} = \textsc{Cls} \times \textsc{Mthd} \rightarrow \mathrm{P} \\
\mathbb{D} \in & \textsc{Dep} = \textsc{Cls} \rightarrow \mathrm{P} & \mathbb{B} \in & \textsc{Pre} = \textsc{Cls} \times \textsc{Mthd} \times \mathrm{R} \rightarrow \mathrm{P} \\
\mathbb{E} \in & \textsc{End} = \textsc{Cls} \times \textsc{Mthd} \rightarrow \mathrm{P} & \mathbb{C} \in & \textsc{Cll} = \textsc{Cls} \times \textsc{Mthd} \times \textsc{Cls} \rightarrow \mathrm{R} \\
\mathbb{U} \in & \textsc{Upd} = \textsc{Cls} \times \textsc{Mthd} \times \textsc{Cls} \times \textsc{Mthd} \rightarrow \mathrm{R} & &
\end{array}$$

To describe a verification technique applied to a program, we write the application of the components to class, method names, *etc.*, as $\mathbb{X}_{c,m}$, $\mathbb{V}_{c,m}$, $\mathbb{D}_c$, $\mathbb{B}_{c,m,\mathbb{r}}$, $\mathbb{E}_{c,m}$, $\mathbb{U}_{c,m,c'}$, $\mathbb{C}_{c,m,c',m'}$. The meaning of these components is:

$\mathbb{X}_{c,m}$: the property expected to be valid at the beginning and end of the body of method m in class c. The parameters c and m allow a verification technique to expect different invariants in the visible states of different methods. For instance, JML's helper methods [17] do not expect any invariants to hold.

$\mathbb{V}_{c,m}$: the property vulnerable to method m of class c, that is, the property whose validity may be broken while control is inside m. The parameters c and m allow a verification technique to require that invariants of certain classes (for instance, c's subclasses) are not vulnerable.

$\mathbb{D}_c$: the property that may depend on fields declared in class c. The parameter c can be used, for instance, to prevent invariants from depending on fields declared in c's superclasses [16, 27].

$\mathbb{B}_{c,m,\mathbb{r}}$: the property whose validity has to be proven before calling a method on a receiver in region $\mathbb{r}$ from the execution of a method m in class c. The parameters allow proof obligations to depend on the calling method and the ownership relation between the caller and the callee.

$\mathbb{E}_{c,m}$: the property whose validity has to be proven at the end of method m in class c. The parameters allow a technique to require different proofs for different methods, *e.g.*, to exclude subclass invariants.

$\mathbb{U}_{c,m,c'}$: the region of allowed receivers for an update of a field in class c', within the body of method m in class c. The parameters allow a technique, for instance, to prevent field updates within pure methods.

$\mathbb{C}_{c,m,c',m'}$: the region of allowed receivers for a call to method m' of class c', within the body of method m of class c. The parameters allow a technique to permit calls depending on attributes (*e.g.*, purity or effect specifications) of the caller and the callee.

The class and method identifiers used as parameters to our components can be extracted from an environment Γ or a stack frame σ in the obvious way. Thus, for $\Gamma = (c, m, \_)$ or for $\sigma = (\iota, \_, c, m)$, we use $\mathbb{X}_\Gamma$ and $\mathbb{X}_\sigma$ as shorthands for $\mathbb{X}_{c,m}$; we also use $\mathbb{B}_{\Gamma,\mathbb{r}}$ and $\mathbb{B}_{\sigma,\mathbb{r}}$ as shorthands for $\mathbb{B}_{c,m,\mathbb{r}}$.

Well-Verified Programs. The judgement $\Gamma \vdash_{\mathcal{V}} e$ expresses that expression e is *well-verified* according to verification technique $\mathcal{V}$. It is defined in Fig. 5.

$$\text{(vs-null)}\ \frac{}{\Gamma \vdash_{\mathcal{V}} \mathsf{null}} \qquad \text{(vs-var)}\ \frac{}{\Gamma \vdash_{\mathcal{V}} x} \qquad \text{(vs-this)}\ \frac{}{\Gamma \vdash_{\mathcal{V}} \mathsf{this}} \qquad \text{(vs-new)}\ \frac{}{\Gamma \vdash_{\mathcal{V}} \mathsf{new}\, t} \qquad \text{(vs-fld)}\ \frac{\Gamma \vdash_{\mathcal{V}} e}{\Gamma \vdash_{\mathcal{V}} e.f}$$

$$\text{(vs-ass)}\ \frac{\begin{array}{l}\Gamma \vdash e : \mathbb{r}\, c' \quad \mathcal{F}(c', f) = \_, c \\ \mathbb{r} \sqsubseteq \mathbb{U}_{\Gamma,c} \quad \Gamma \vdash_{\mathcal{V}} e \quad \Gamma \vdash_{\mathcal{V}} e'\end{array}}{\Gamma \vdash_{\mathcal{V}} e.f = e'}$$

$$\text{(vs-call)}\ \frac{\begin{array}{l}\Gamma \vdash e : \mathbb{r}\, c' \quad \mathcal{B}(c', m) = \_, c \\ \mathbb{r} \sqsubseteq \mathbb{C}_{\Gamma,c,m} \quad \Gamma \vdash_{\mathcal{V}} e \quad \Gamma \vdash_{\mathcal{V}} e'\end{array}}{\Gamma \vdash_{\mathcal{V}} e.m(e'\ \mathsf{prv}\, \mathbb{B}_{\Gamma,\mathbb{r}})} \qquad \text{(vs-class)}\ \frac{\left.\begin{array}{l}\mathcal{B}(c, m) = e, c \\ \mathcal{M}(c, m) = t, t'\end{array}\right\} \Rightarrow \left\{\begin{array}{l} e = e'\ \mathsf{prv}\, \mathbb{E}_{c,m} \\ c, m, t \vdash_{\mathcal{V}} e'\end{array}\right.}{\vdash_{\mathcal{V}} c}$$

Fig. 5. Well-verified source expressions and classes.

The first five rules express that literals, variable lookup, object creation, and field lookup do not require proofs. The receiver of a field update must fall into $\mathbb{U}$ (vs-ass). The receiver of a call must fall into $\mathbb{C}$ (vs-call). Moreover, we require the proof of $\mathbb{B}$ before a call. Finally, a class is well-verified if the body of each of its methods is well-verified and ends with a proof obligation for $\mathbb{E}$ (vs-class). Note that we use the type judgement $\Gamma \vdash e : t$ without defining it; the definition is given by the underlying programming language, not by our framework.

Fig. 9 in App. A defines the judgement $h \vdash_{\mathcal{V}} e_r$ for verified runtime expressions. The rules correspond to those from Fig. 5, with the addition of rules for values and nested calls.

A program Π is well-verified *w.r.t.* $\mathcal{V}$, denoted as $\vdash_{\mathcal{V}} \Pi$, iff *(1)* all classes are well-verified and *(2)* all class invariants respect the dependency restrictions dictated by $\mathbb{D}$. That is, the invariant of an object ι' declared in a class c' will be preserved by an update of a field of an object of class c if it is not within $\mathbb{D}_c$.

Definition 2 $\vdash_{\mathcal{V}} \Pi \ \Leftrightarrow$

(1) $\forall c \in \Pi.\ \vdash_{\mathcal{V}} c$

(2) $\mathcal{F}(cls(h, \iota), f) = \_, c,\ (\iota', c') \notin [\![\mathbb{D}_c]\!]_{h,\iota},\ h \models \iota', c' \ \Rightarrow\ upd(h, \iota, f, v) \models \iota', c'$

Valid States. The properties $\mathbb{X}$ and $\mathbb{X} \setminus \mathbb{V}$ characterise the invariants that are expected to hold in the visible states and between visible states of the current method execution, respectively. That is, they reflect the local knowledge of the current method, but do not describe globally all the invariants that need to hold in a given state.

For any state with heap h and execution stack $\overline{\sigma}$, the function $vi(\overline{\sigma}, h)$ yields the set of *valid invariants*, that is, invariants that are expected to hold :

$$vi(\overline{\sigma}, h) = \begin{cases} \emptyset & \text{if } \overline{\sigma} = \epsilon \\ (vi(\overline{\sigma_1}, h) \cup [\![\mathbb{X}_\sigma]\!]_{h,\sigma}) \backslash [\![\mathbb{V}_\sigma]\!]_{h,\sigma} & \text{if } \overline{\sigma} = \overline{\sigma_1} \cdot \sigma \end{cases}$$

The call stack is empty at the beginning of program execution, at which point we expect the heap to be empty. For each additional stack frame σ, the corresponding method m may assume $\mathbb{X}_\sigma$ at the beginning of the call, therefore we add $[\![\mathbb{X}_\sigma]\!]_{h,\sigma}$ to the valid invariants. The method may break $\mathbb{V}_\sigma$ during the call, and so we remove $[\![\mathbb{V}_\sigma]\!]_{h,\sigma}$ from the valid invariants.

A state with heap h and stack $\overline{\sigma}$ is *valid* iff:

(1) $\overline{\sigma}$ is a valid stack, denoted by $h \vdash_{\mathcal{V}} \overline{\sigma}$ (Def. 12 in App. A), and meaning that the receivers of consecutive method calls are within the respective $\mathbb{C}$ regions.
(2) The valid invariants $vi(\overline{\sigma}, h)$ hold.
(3) If execution is in a visible state with σ as the topmost frame of $\overline{\sigma}$, then the expected invariants $\mathbb{X}_\sigma$ hold additionally.

These properties are formalised in Def. 3. A state is determined by a heap h and a runtime expression e_r; the stack is extracted from e_r using function *stack*, given by Def. 13 in App. A.

Definition 3 *A state with heap h and runtime expression e_r is valid for a verification technique $\mathcal{V}$, $e_r \models_{\mathcal{V}} h$, iff:*

$$\text{(1)}\ h \vdash_{\mathcal{V}} stack(e_r) \qquad \text{(2)}\ h \models vi(stack(e_r), h)$$
$$\text{(3)}\ e_r = F[\sigma \cdot \mathsf{call}\, e] \ \text{ or } \ e_r = F[\sigma \cdot \mathsf{ret}\, v] \ \Rightarrow \ h \models \mathbb{X}_\sigma,\ \sigma$$

Soundness. A verification technique is *sound* if verified programs only produce valid states and do not throw fatal exceptions. More precisely, a verification technique $\mathcal{V}$ is sound for a programming language PL iff for all well-formed and verified programs $\Pi \in PL$, any well-typed and verified runtime expression e_r executed in a valid state reduces to another verified expression e_r' with a resulting valid state. Note that a verified e_r' contains no $\mathsf{fatalExc}$ (see Fig. 9).

Well-formedness of program Π is denoted by $\vdash_{\mathbf{wf}} \Pi$ (Def. 14, App. A). Well-typedness of runtime expression e_r is denoted by $h \vdash e_r : t$ and required as part of a sound type system in Def. 11, App. A. These requirement permits separation of concerns, whereby we can formally define verification technique soundness *in isolation*, assuming program well-formedness and a sound type system.

Definition 4 *A verification technique $\mathcal{V}$ is sound for a programming language PL iff for all programs $\Pi \in PL$:*

$$\left.\begin{array}{l} \vdash_{\mathbf{wf}} \Pi, \quad h \vdash e_r : \_, \quad \vdash_{\mathcal{V}} \Pi, \quad e_r \models_{\mathcal{V}} h, \\ h \vdash_{\mathcal{V}} e_r, \quad e_r,\, h \longrightarrow e_r',\, h' \end{array}\right\} \Rightarrow \ e_r' \models_{\mathcal{V}} h', \quad h' \vdash_{\mathcal{V}} e_r'$$

5 Instantiations

In our earlier paper [7], we discuss six verification techniques from the literature in terms of our framework, namely those by Poetzsch-Heffter [31], Huizing &

Kuiper [14], Leavens & Müller [16], Müller *et al.* [27], and Lu *et al.* [23]. In this paper we concentrate on the techniques based on heap topologies [27, 23], because those benefit most from the formalisation in our framework.

Müller *et al.* [27] present two techniques for multi-object invariants, called ownership technique and visibility technique (OT and VT for short), which use the hierarchic heap topology enforced by Universe types [6]. Their distinctive features are: (1) Expected and vulnerable invariants are specified per class. (2) Invariant restrictions take into account subclassing (thereby addressing the subclass challenge). (3) Proof obligations are required before calls (thereby addressing the call-back challenge) and at the end of calls. (4) Program restrictions are uniform for all methods[3], and are based on the relative object placement in the hierarchy.

Lu *et al.* [23] define *Oval*, a verification technique based on ownership types, which support owner parameters for classes [5], thus permitting a more precise description of the heap topology. The distinctive features of *Oval* are: (1) Expected and vulnerable invariants are specific to every method in every class through the notion of *contracts*. (2) Invariant restrictions do not take subclassing into account. (3) Proof obligations are only imposed at the end of calls. (4) To address the call-back challenge, calls are subject to "subcontracting", a requirement that guarantees that the expected and vulnerable invariants of the callee are within those of the caller.

OT, VT, and *Oval* are discussed in more detail in our companion report [8]. In the remainder of this section, we introduce these techniques and summarise them in Fig. 6. We explain the notation from Fig. 6 informally, and define it formally in the appendix. This section (without the appendix) gives an overall intuition, aimed at the reader who is not interested in all of the formal details.

To sharpen our discussion *w.r.t.* structured heaps, we will be adding annotations to the example from Fig. 1, to obtain a topology where the Person p owns the Account a and the DebitCard d.

5.1 Instantiation for *OT* and *VT*

Universe types associate reference types with *Universe modifiers*, which specify ownership relative to the current object. The modifier **rep** expresses that an object is owned by the current object; **peer** expresses that an object has the same owner as the current object; **any** expresses that an object may have any owner. Fig. 7 shows the Universe modifiers for our example from Fig. 1, which allow one to apply OT and VT.

To address the subclass challenge, OT and VT both forbid **rep** fields f and g declared in different classes c_f and c_g of the same object o to reference the same object. This *subclass separation* can be formalised in an ownership model where each object is owned by an object-class pair (see [18] for details).

[3] However, both OT and VT have special rules for pure (side-effect free) methods. We ignore pure methods here, but refer the interested reader to [7].

	Müller *et al.* (*OT*)	Müller *et al.* (*VT*)	Lu *et al.*(*Oval*)
$\mathbb{X}_{c,m}$	$\mathsf{own}\,;\mathsf{rep}^{+}$	$\mathsf{own}\,;\mathsf{rep}^{+}$	$\mathsf{I}\,;\mathsf{rep}^{*}$
$\mathbb{V}_{c,m}$	$\mathsf{super}\langle c\rangle \sqcup \mathsf{own}^{+}$	$\mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^{+}$	$\mathsf{E}\,;\mathsf{own}^{*}$
$\mathbb{D}_{c}$	$\mathsf{self}\langle c\rangle \sqcup \mathsf{own}^{+}$	$\mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^{+}$	$\mathsf{self}\,;\mathsf{own}^{*}$
$\mathbb{B}_{c,m,\mathbb{r}}$	$\mathsf{super}\langle c\rangle$ if $\mathsf{intrsPeer}(\mathbb{r})$ emp otherwise	$\mathsf{peer}\langle c\rangle$ if $\mathsf{intrsPeer}(\mathbb{r})$ emp otherwise	emp
$\mathbb{E}_{c,m}$	$\mathsf{super}\langle c\rangle$	$\mathsf{peer}\langle c\rangle$	self if $\mathsf{I}=\mathsf{E}$ emp otherwise
$\mathbb{U}_{c,m,c'}$	self	peer	self if $\mathsf{I}=\mathsf{E}$ emp otherwise
$\mathbb{C}_{c,m,c',m'}$	$\mathsf{rep}\langle c\rangle \sqcup \mathsf{peer}$	$\mathsf{rep}\langle c\rangle \sqcup \mathsf{peer}$	$\bigsqcup_{\mathbb{r},\ \text{with}\ \mathsf{SC}(\mathsf{I},\mathsf{E},\mathsf{I}',\mathsf{E}',\mathcal{O}_{\mathbb{r},c})} \mathbb{r}$

Fig. 6. Components of verification techniques. For *Oval*, $\mathcal{O}_{\mathbb{r},c}$ is the owner of $\mathbb{r}$; we use shorthands $\mathsf{I} = \mathsf{I}(c, m)$, and $\mathsf{E} = \mathsf{E}(c, m)$, and $\mathsf{I}' = \mathbb{r}\,; \mathsf{I}(c', m')$, and $\mathsf{E}' = \mathbb{r}\,; \mathsf{E}(c', m')$.

```
class Account {              class Person {              class DebitCard {
   peer DebitCard card;         rep Account account;        peer Account acc;
   any Person holder;           ...                         ...
   ...                       }                           }
}
```

Fig. 7. Universe modifiers for the Account example from Fig. 1.

Regions and Properties. For *OT* and *VT*, we define the sets of regions and properties to be:

$$\mathbb{r} \in \mathrm{R} \ ::= \ \mathsf{emp} \mid \mathsf{self} \mid \mathsf{rep}\langle c\rangle \mid \mathsf{peer} \mid \mathsf{any} \mid \mathbb{r} \sqcup \mathbb{r}$$
$$\mathbb{p} \in \mathrm{P} \ ::= \ \mathsf{emp} \mid \mathsf{self}\langle c\rangle \mid \mathsf{super}\langle c\rangle \mid \mathsf{peer}\langle c\rangle \mid \mathsf{rep} \mid \mathsf{own} \mid \mathsf{rep}^{+} \mid \mathsf{own}^{+} \mid \mathbb{p} \sqcup \mathbb{p} \mid \mathbb{p}; \mathbb{p}$$

In our framework, Universe modifiers intuitively correspond to regions, since they describe areas of the heap. For example, peer describes all objects which share the owner (object-class pair) with the current object. However, because of the subclass separation described above, it is useful to employ richer regions of the form $\mathsf{rep}\langle c\rangle$, describing all objects owned by the current object *and* class c. For regions (and properties) we also include the "union" of two regions (properties). The predicate $\mathsf{intrsPeer}(\mathbb{r})$ checks whether a region intersects the peer region.

For properties, $\mathsf{self}\langle c\rangle$ represents the singleton set containing a pair of the current object with the class c. The property $\mathsf{super}\langle c\rangle$ represents the set of pairs of the current object with all its classes that are superclasses of c. The property $\mathsf{peer}\langle c\rangle$ represents all the objects (paired with their classes) that share the owner with the current object, provided their class is visible in c. There are also properties to describe the invariants of an object's owned objects, its owner, its transitively owned objects, and its transitive owners. A property of the form $\mathbb{p}_1; \mathbb{p}_2$ denotes a composition of properties, which behaves similarly to function composition when interpreted. The formal definitions of the interpretations of these regions and properties can be found in App. B.

Ownership Technique. As shown in Fig. 6, *OT* requires that in visible states, all objects owned by the owner of **this** must satisfy their invariants ($\mathbb{X}$).

Invariants are allowed to depend on fields of the object itself (at the current class), as in I1 in Fig. 1, and all its **rep** objects, as in I2. Other client invariants such as I4 and I5) and subclass invariants that depend on inherited fields (such as I3) are not permitted. Therefore, a field update potentially affects the invariants of the modified object and of all its (transitive) owners ($\mathbb{D}$).

A method may update fields of **this** ($\mathbb{U}$). Since an updated field is declared in the enclosing class or a superclass, the invariants potentially affected by the update are those of **this** (for the enclosing class and its superclasses, which addresses the subclass challenge) as well as the invariants of the (transitive) owners of **this** ($\mathbb{V}$).

OT handles multi-object invariants by allowing invariants to depend on fields of owned objects ($\mathbb{D}$). Therefore, methods may break the invariants of the transitive owners of **this** ($\mathbb{V}$). For example, the invariant I2 of Person (Fig. 1) is legal only because account is a **rep** field (Fig. 7). Account's method withdraw need not preserve Person's invariant. This is reflected by the definition of $\mathbb{E}$: only the invariants of **this** are proven at the end of the method, while those of the transitive owners may remain broken; it is the responsibility of the owners to re-establish them, which addresses the multi-object challenge. As an example, the method spend has to re-establish Person's invariant after the call to account.withdraw.

Since the invariants of the owners of **this** might not hold, *OT* disallows calls on references other than **rep** and **peer** references ($\mathbb{C}$). For instance, the call holder.notify() in method sendReport is not permitted because holder is in an ancestor ownership context.

The proof obligations for method calls ($\mathbb{B}$) must cover those invariants expected by the callee that are vulnerable to the caller. This intersection contains the invariant of the caller, if the caller and the callee are peers because the callee might call back; it is otherwise empty (**rep**s cannot callback their owners).

Visibility Technique. *VT* relaxes the restrictions of *OT* in two ways. First, it permits invariants of a class c to depend on fields of peer objects, provided that these invariants are visible in c ($\mathbb{D}$). Thus, *VT* can handle multi-object structures that are not organised hierarchically. For instance, in addition to the invariants permitted by *OT*, *VT* permits invariants I4 and I5 in Fig. 1. Visibility is transitive, thus, the invariant must also be visible wherever fields of c are updated. Second, *VT* permits field updates on peers of **this** ($\mathbb{U}$).

These relaxations make more invariants vulnerable. Therefore, $\mathbb{V}$ includes additionally the invariants of the peers of **this**. This addition is also reflected in the proof obligations before peer calls ($\mathbb{B}$) and before the end of a method ($\mathbb{E}$). For instance, method withdraw must be proven to preserve the invariant of the associated DebitCard, which does not in general succeed in our example.

5.2 Instantiation for *Oval*

Fig. 8 shows our example in *Oval* using ownership parameters [5] to describe heap topologies. The ownership parameter o denotes the owner of the current object;

p denotes the owner of o and specifies the position of holder in the hierarchy, more precisely than the **any** modifier in Universe types.

```
class Account[o,p] {                          class Person[o] {
  DebitCard⟨o⟩ card;                            Account⟨this⟩ account;
  Person⟨p⟩ holder;                             ...
  ...                                           void spend(int amount)⟨this,this⟩
  void withdraw(int amount)⟨this,this⟩            { account.withdraw(amount); }
    { ... }                                     void notify ()⟨bot,top⟩
  void sendReport()⟨bot,p⟩                        { ... }
    { ... }                                   }
}
```

Fig. 8. Ownership parameters and method contracts in *Oval*.

Method Contracts. Ownership parameters are also used to describe expected and vulnerable invariants, which are specific to each method. Every *Oval* program extends method signatures with a contract $\langle \mathsf{I}, \mathsf{E}\rangle$: the expected invariants at visible states ($\mathbb{X}$) are the invariants of the object characterised by I and all objects transitively owned by this object; the vulnerable invariants ($\mathbb{V}$) are the object at E and its transitive owners. These properties are syntactically characterised by Ls in the code (and Ks in typing rules), where:

$$L ::= \mathsf{top} \mid \mathsf{bot} \mid \mathsf{this} \mid X \qquad\qquad K ::= L \mid K ; \mathsf{rep}$$

and where X stands for the class' owner parameters.[4] An ordering $L \preceq L'$ is defined, expressing that at runtime the object denoted by L will be transitively owned by the object denoted by L'. This is used to formally specify various restrictions in the technique, for example that for all method contracts, $\mathsf{I} \preceq \mathsf{E}$ must hold.

In class Account (Fig. 8), withdraw() expects the current object and the objects it transitively owns to be valid (I=**this**) and, during execution, this method may invalidate the current object and its transitive owners (E=**this**). The contract of sendReport() does not expect any objects to be valid at visible states (I=**bot**) but may violate object p and its transitive owners (E=p).

Subcontracting. Call-backs are handled via *subcontracting*, which is defined using the order $L \preceq L'$.To interpret *Oval*'s subcontracting in our framework, we use $\mathsf{SC}(\mathsf{I}, \mathsf{E}, \mathsf{I}', \mathsf{E}', K)$, which holds iff:

$$\mathsf{I} \prec \mathsf{E} \Rightarrow \mathsf{I}' \preceq \mathsf{I} \qquad \mathsf{I} = \mathsf{E} \Rightarrow \mathsf{I}' \prec \mathsf{I} \qquad \mathsf{I}' \prec \mathsf{E}' \Rightarrow \mathsf{E} \preceq \mathsf{E}' \qquad \mathsf{I}' = \mathsf{E}' \Rightarrow \mathsf{E} \preceq K$$

[4] We discuss a slightly simplified version of *Oval*, where we omit the existential owner parameter '$*$', and *non-rep* fields, a refinement whereby only the current object's owners depend on such fields. Both enhance the expressiveness of the language, but are not central to our analysis.

where I, E characterise the caller, I', E' characterise the callee, and K stands for the callee's owner. The first two requirements ensure that the caller guarantees the invariant expected by the callee. The other two conditions ensure that the invariants vulnerable to the callee are also vulnerable to the caller. For instance, the call holder. notify () in method sendReport satisfies subcontracting because caller and callee do not expect any invariants, and the callee has no vulnerable invariants. In particular, the receiver of a call may be owned by any of the owners of the current receiver, provided that subcontracting is respected ($\mathbb{C}$).

Given that $\mathsf{I} \preceq \mathsf{E}$ for all well-formed methods, and that $\mathbb{B}_{c,m,\mathbb{r}}$ =emp, the first two requirements of subcontracting exactly give *(S1)*, while the latter two exactly give *(S3)* from Def. 5 in the next section – more in [8].

Regions and Properties. To express *Oval* in our framework, we define regions and properties as follows (see App. B for their interpretations):

$$\mathbb{r} \in \mathrm{R} ::= \mathsf{emp} \mid \mathsf{self} \mid c\langle\overline{K}\rangle \mid \mathbb{r} \sqcup \mathbb{r} \qquad \mathbb{p} \in \mathrm{P} ::= \mathsf{emp} \mid \mathsf{self} \mid K \mid K;\mathsf{rep}^* \mid K;\mathsf{own}^*$$

As already stated, expected and vulnerable properties depend on the contract of the method and express $\mathbb{X}$ as $\mathsf{I};\mathsf{rep}^*$ and $\mathbb{V}$ as $\mathsf{E};\mathsf{own}^*$ (see Fig. 6). Similarly to *OT*, invariant dependencies are restricted to an object and the objects it transitively owns ($\mathbb{D}$). Therefore, I1 and I4 are legal, as well as I3, which depends on an inherited field. *Oval* imposes a restriction on contracts that the expected and vulnerable invariants of every method intersect at most at this. Consequently, at the end of a method, one has to prove the invariant of the current receiver, if $\mathsf{I} = \mathsf{E} = \mathsf{this}$, and nothing otherwise ($\mathbb{E}$). In the former case, the method is allowed to update fields of its receiver; no updates are allowed otherwise ($\mathbb{U}$). Therefore, spend and withdraw are the only methods in our example that are allowed to make field updates. *Oval* does not impose proof obligations on method calls ($\mathbb{B}$ is empty), but addresses the call-back challenge through subcontracting. Therefore, call-backs are safe because the callee cannot expect invariants that are temporarily broken. With the existing contracts in Fig. 8, subcontracting permits spend to call account.withdraw(), and withdraw to call **this**.sendReport(), and also sendReport to call holder. notify (). The last two subcalls may potentially lead to callbacks, but are safe because the contracts of sendReport and notify do not expect the receiver to be in a valid state (I=**bot**).

Subclassing and Subcontracting. Oval also requires subcontracting between a superclass method and an overriding subclass method. As we discuss later, this does not guarantee soundness [22], and we found a counterexample (*cf.* Sec. 6). Therefore, we require that a subclass expects no more than the superclass, and vice versa for vulnerable invariants, and that if an expected invariant in the superclass is vulnerable in the subclass, then it must also be expected in the subclass:[5]

$$\mathsf{I}' \preceq \mathsf{I} \preceq \mathsf{E} \preceq \mathsf{E}' \qquad \mathsf{I} = \mathsf{E}' \;\Rightarrow\; \mathsf{I}' = \mathsf{E}'$$

[5] Note, that we had erroneously omitted the latter requirement in [7].

where I, E, I', E' characterise the superclass, resp. subclass, method. This requirement gives exactly *(S5)* from Def. 5. It allows $I' = I = E = E'$ which is forbidden in Oval. We refer to the verification technique with the above requirement for method overriding as *Oval′*.

5.3 Summary

In spite of differences in, *e.g.,* the underlying type systems and the logics used, our framework allows us to extract comparable information about these three techniques. We summarise here the commonalities and differences in the results.

1. *Invariant semantics:* In *OT* and *VT*, the invariants expected at the beginning of withdraw are I1, I2, and I3 for the receiver, as well as I5 for the associated DebitCard (which is a **peer**). For withdraw in *Oval*, I=this, therefore the expected invariants are I1, I2, and I3 for the receiver.
2. *Invariant restrictions:* Invariants I2 and I5 are illegal in *OT* and *Oval*, while they are legal in *VT* (which allows invariants to depend on the fields of **peer**s). Conversely, I3 is illegal in *OT* and *VT* (it mentions a field from a superclass), while it is legal in *Oval*.
3. *Proof obligations:* In *OT*, before the call to **this**.sendReport() and at the end of the body of withdraw, we have to establish I1 and I2 for the receiver. In addition to these, in *VT* we have to establish I5 for the debit card. In *Oval*, the same invariants as for *OT* have to be proven, but only at the end of the method because call-backs are handled through subcontracting. In addition, I3 is required.[6] In all three techniques, withdraw is permitted to leave the invariant I4 of the owning Person object broken. It has to be re-established by the calling Person method.
4. *Program restrictions:* *OT* and *VT* forbid the call holder.notify() (**rep**s cannot call their owners), while *Oval* allows it. On the other hand, if method sendReport required an invariant of its receiver (for instance, to ensure that holder is non-null), then *Oval* would prevent method withdraw from calling it, even though the invariants of the receiver might hold at the time of the call. The proof obligations before calls in *OT* and *VT* would make such a call legal.

6 Well-Structured Verification Techniques

We now identify conditions on the components of a verification technique that are sufficient for soundness, state a general soundness theorem, and discuss soundness of the techniques presented in Sec. 5

Definition 5 *A verification technique is* well-structured *if, for all programs in the programming language:*

[6] This means that verification of a class requires knowledge of a subclass. The *Oval* developers plan to solve this modularity problem by requiring that any inherited method has to be re-verified in the subclass [22].

(S1) $\mathbb{r} \sqsubseteq \mathbb{C}_{c,m,c'm'} \Rightarrow (\mathbb{r} \triangleright \mathbb{X}_{c',m'}) \setminus (\mathbb{X}_{c,m} \setminus \mathbb{V}_{c,m}) \ \subseteq \ \mathbb{B}_{c,m,\mathbb{r}}$
(S2) $\mathbb{V}_{c,m} \cap \mathbb{X}_{c,m} \subseteq \mathbb{E}_{c,m}$
(S3) $\mathbb{C}_{c,m,c',m'} \triangleright (\mathbb{V}_{c',m'} \setminus \mathbb{E}_{c',m'}) \ \subseteq \ \mathbb{V}_{c,m}$
(S4) $\mathbb{U}_{c,m,c'} \triangleright \mathbb{D}_{c'} \ \subseteq \ \mathbb{V}_{c,m}$
(S5) $c' <: c \ \Rightarrow \ \mathbb{X}_{c',m} \subseteq \mathbb{X}_{c,m} \ \wedge \ \mathbb{V}_{c',m} \setminus \mathbb{E}_{c',m} \subseteq \mathbb{V}_{c,m} \setminus \mathbb{E}_{c,m}$

In the above, the set theoretic symbols have the obvious interpretation in the domain of properties. For example *(S2)* is short for $\forall h, \iota : \ [\![\mathbb{V}_{c,m}]\!]_{h,\iota} \cap ([\![\mathbb{X}_c]\!]_{h,\iota} \subseteq [\![\mathbb{E}_{c,m}]\!]_{h,\iota}$. We use *viewpoint adaptation* $\mathbb{r} \triangleright \mathbb{p}$, defined as:

$$[\![\mathbb{r} \triangleright \mathbb{p}]\!]_{h,\iota} = \bigcup\nolimits_{\iota' \in [\![\mathbb{r}]\!]_{h,\iota}} [\![\mathbb{p}]\!]_{h,\iota'}$$

meaning that the interpretation of a viewpoint-adapted property $\mathbb{r} \triangleright \mathbb{p}$ *w.r.t.* an address ι is equal to the union of the interpretations of $\mathbb{p}$ *w.r.t.* each object in the interpretation of $\mathbb{r}$.

The first two conditions relate proof obligations with expected invariants. *(S1)* ensures for a call within the permitted region that the expected invariants of the callee $(\mathbb{r} \triangleright \mathbb{X}_{c',m'})$ minus the invariants that hold throughout the calling method $(\mathbb{X}_{c,m} \setminus \mathbb{V}_{c,m})$ are included in the proof obligation for the call $(\mathbb{B}_{c,m,\mathbb{r}})$. *(S2)* ensures that the invariants that were broken during the execution of a method, but which are required to hold again at the end of the method $(\mathbb{V}_{c,m} \cap \mathbb{X}_{c,m})$ are included in the proof obligation at the end of the method $(\mathbb{E}_{c,m})$.

The third and fourth condition ensure that invariants that are broken by a method m of class c are actually in its vulnerable set. Condition *(S3)* deals with calls and therefore uses viewpoint adaptation for call regions $(\mathbb{C}_{c,m,c',m'} \triangleright \ldots)$. It restricts the invariants that may be broken by the callee method m', but are not re-established by the callee through $\mathbb{E}$. These invariants must be included in the vulnerable invariants of the caller. Condition *(S4)* ensures for field updates within the permitted region that the invariants broken by updating a field of class c' are included in the vulnerable invariants of the enclosing method, m.

Finally, *(S5)* establishes conditions for subclasses. An overriding method m in a subclass c may expect fewer invariants than the overridden m in superclass c'. Moreover, the subclass method must leave less invariants broken than the superclass method.

Note that the five soundness conditions presented here are slightly weaker than those in the previous version of this work [7]. [7]

Soundness Results. The five conditions from Def. 5 guarantee soundness of a verification technique (Def. 4), provided that the programming language has a sound type system (see Def. 15 in App. A).

[7] Namely, *(S3)* and *(S5)* are weaker, and thus less restrictive, here. In [7], instead of *(S3)* we required the stronger version $\mathbb{C}_{c,m,c',m'} \triangleright \ (\mathbb{V}_{c',m'} \setminus \mathbb{X}_{c',m'}) \subseteq \mathbb{V}_{c,m}$, and a similarly stronger version for *(S5)*. However, the two versions are equivalent when $\mathbb{E}_{c,m}$ is the minimal set allowed by *(S2)*, *i.e.,* when $\mathbb{E}_{c,m} = \mathbb{V}_{c,m} \cap \mathbb{X}_{c,m}$ for all c and m. In all techniques presented here, $\mathbb{E}_{c,m}$ is minimal in the above sense.

Theorem 6 *A well-structured verification technique, built on top of a programming language with a sound type system, is sound.*

This theorem is one of our main results. It reduces the complex task of proving soundness of a verification technique to checking five fairly simple conditions.

Unsoundness of Oval. The original *Oval* proposal [23] is unsound because it requires subcontracting for method overriding. As we said in the previous section, subcontracting corresponds to our *(S1)* and *(S3)*. This gives, for $c' <: c$, the requirements that $\mathbb{X}_{c',m'} \subseteq \mathbb{X}_{c,m} \setminus \mathbb{V}_{c,m}$, and $\mathbb{V}_{c',m'} \setminus \mathbb{E}_{c',m'} \subseteq \mathbb{V}_{c,m}$, which do not imply *(S5)*. We were alerted by this discrepancy, and using only the $\mathbb{X}$, $\mathbb{E}$ and $\mathbb{V}$ components (no type system properties, nor any other component), we constructed the following counterexample.

```
class D[o] {                                  class C1[o]{
                                                 void mm() <this,this> {...}
   C1<this> c = new C2<this>();               }
   void m() <this,o> { c.mm() }               class C2[o] extends C1<o> {
                                                 void mm() <bot,this> {...}
}                                             }
```

The call c.mm() is checked using the contract of C1::mm; it expects the callee to re-establish the invariant of the receiver (c), and is type correct. However, the body of C2::mm may break the receiver's invariants, but has no proof obligations ($\mathbb{E}_{\mathsf{C2},\mathsf{mm}} = \mathsf{emp}$). Thus, the call c.mm() might break the invariants of c, thus breaking the contract of m. The reason for this problem is, that the—initially appealing—parallel between subcontracting and method overriding does not hold. The authors confirmed our findings [22].

Soundness of the Presented Techniques.

Theorem 7 *The verification techniques OT, VT, and Oval′ are well-structured.*

Corollary 8 *The verification techniques OT, VT, and Oval′ are sound.*

Our proof of Corollary 8 confirmed soundness claims from the literature. We found that the semi-formal arguments supporting the original soundness claims at times missed crucial steps. For instance, the soundness proofs for *OT* and *VT* [27] do not mention any condition relating to *(S3)* of Def. 5; in our formal proof, *(S3)* was vital to determine what invariants still hold after a method returns. We relegate proofs of the theorems to the companion report [8].

7 Related Work

Object invariants trace back to Hoare's implementation invariants [12] and monitor invariants [13]. They were popularised in object-oriented programming by Meyer [24]. Their work, as well as other early work on object invariants [20, 21] did not address the three challenges described in the introduction. Since they were not formalised, it is difficult to understand the exact requirements and soundness arguments (see [27] for a discussion). However, once the requirements are clear, a formalisation within our framework seems straightforward.

The idea of regions and properties is inspired from type and effects systems [33], which have been extremely widely applied, *e.g.,* to support race-free programs and atomicity [10].

The verification techniques based on the Boogie methodology [1, 3, 18, 19] do not use a visible state semantics. Instead, each method specifies in its precondition which invariants it requires. Extending our framework to Spec# requires two changes. First, even though Spec# permits methods to specify explicitly which invariants they require, the default is to require the invariants of its arguments and all their peer objects. These defaults can be modelled in our framework by allowing method-specific properties $\mathbb{X}$. Second, Spec# checks invariants at the end of expose blocks instead of the end of method bodies. Expose blocks can easily be added to our formalism.

In separation logic [15, 32], object invariants are generally not as important as in other verification techniques. Instead, predicates specifying consistency criteria can be assumed/proven at *any* point in a program [28]. Abstract predicate families [29] allow one to do so without violating abstraction and information hiding. Parkinson and Bierman [30] show how to address the subclass challenge with abstract predicates. Their work as well as Chin *et al.*'s [4] allow programmers to specify which invariants a method expects and preserves, and do not require subclasses to maintain inherited invariants. The general predicates of separation logic provide more flexibility than can be expressed by our framework.

We know of only one technique based on visible states that cannot be directly expressed in our framework: Middelkoop *et al.* [26] use proof obligations that refer to the heap of the pre-state of a method execution. To formalise this technique, we have to generalise our proof obligations to take two properties; one for the pre-state heap and one for the post-state heap. Since this generality is not needed for any of the other techniques, we omitted a formal treatment in this paper.

Some verification techniques exclude the pre- and post-states of so-called helper methods from the visible states [16, 17]. Helper methods can easily be expressed in our framework by choosing different parameters for helper and non-helper methods. For instance in JML, $\mathbb{X}$, $\mathbb{B}$, and $\mathbb{E}$ are empty for helper methods, because they neither assume nor have to preserve any invariants.

Once established, strong invariants [11] hold throughout program execution. They are especially useful to reason about concurrency and security properties. Our framework can model strong invariants, essentially by preventing them from occurring in $\mathbb{V}$.

Existing techniques for visible state invariants have only limited support for object initialisation. Constructors are prevented from calling methods because the callee method in general requires all invariants to hold, but the invariant of the new object is not yet established. Fähndrich and Xia developed delayed types [9] to control call-backs into objects that are being initialised. Delayed types support strong invariants. Modelling these in our framework is future work.

8 Conclusions

We presented a framework that describes verification techniques for object invariants in terms of seven parameters and separates verification concerns from those of the underlying type system. Our formalism is parametric *w.r.t.* the type system of the programming language and the language used to describe and to prove assumptions. We illustrated the generality of our framework by instantiating it to describe three existing verification techniques. We identified sufficient conditions on the framework parameters that guarantee soundness, and we proved a universal soundness theorem. Our unified framework offers the following important advantages:

1. It allows a simpler understanding and separation of verification concerns. In particular, most of the aspects in which verification techniques differ are distilled in terms of subsets of the parameters of our framework.
2. It facilitates comparisons since relationships between parameters can be expressed at an abstract level (*e.g.,* criteria for well-structuredness in Def. 5), and the interpretations of regions and properties as sets allow formal comparisons of techniques in terms of set operations.
3. It expedites the soundness analysis of verification techniques, since checking the soundness conditions of Def. 5 is significantly simpler than developing soundness proofs from scratch.
4. It captures the design space of *sound* visible states based verification techniques.

We are currently using our framework in developing verification techniques for static methods, and plan to use it to develop further, more flexible, techniques.

Acknowledgements We thank Rustan Leino, Matthew Parkinson, Ronald Middelkoop, John Potter, Yi Lu, as well as the POPL, FOOL and ECOOP referees for their feedback. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

1. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6):27–56, 2004.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, LNCS, pages 49–69. Springer-Verlag, 2005.
3. M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared State. In *MPC*, volume 3125 of *LNCS*, pages 54–84. Springer, 2004.

4. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In *POPL*, pages 87–99. ACM Press, 2008.
5. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, volume 33(10), pages 48–64. ACM Press, 1998.
6. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *JOT*, 4(8):5–32, October 2005.
7. S. Drossopoulou, A. Francalanza, and P. Müller. A unified framework for verification techniques for object invariants. In *FOOL*, 2008.
8. S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summmers. A unified framework for verification techniques for object invariants — ecoop'08 full paper. Available from `http://www.doc.ic.ac.uk/~ajs300m/papers/frameworkFull.pdf`, 2008.
9. M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350. ACM Press, 2007.
10. C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. In *PLDI*, pages 338–349. ACM Press, 2003.
11. R. Hähnle and W. Mostowski. Verification of safety properties in the presence of transactions. In *CASSIS*, volume 3362 of *LNCS*, pages 151–171, 2005.
12. C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.
13. C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.
14. K. Huizing and R. Kuiper. Verification of object-oriented programs using class invariants. In *FASE*, volume 1783 of *LNCS*, pages 208–221. Springer-Verlag, 2000.
15. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26. ACM Press, 2001.
16. G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *ICSE*, pages 385–395. IEEE, 2007.
17. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, February 2007.
18. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.
19. K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *ESOP*, volume 4421 of *LNCS*, pages 316–330. Springer-Verlag, 2007.
20. B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.
21. B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM ToPLAS*, 16(6):1811–1841, 1994.
22. Y. Lu and J. Potter. Soundness of Oval. Priv. Commun., June 2007.
23. Y. Lu, J. Potter, and J. Xue. Object Invariants and Effects. In *ECOOP*, volume 4609 of *LNCS*, pages 202–226. Springer-Verlag, 2007.
24. B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.
25. B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.
26. R. Middelkoop, C. Huizing, R. Kuiper, and E. J. Luit. Invariants for non-hierarchical object structures. *Electr. Notes Theor. Comput. Sci.*, 195:211–229, 2008.
27. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.
28. M. Parkinson. Class invariants: the end of the road? In *International Workshop on Aliasing, Confinement and Ownership*, 2007.

$$\text{(ad-null)}\ \frac{}{h \vdash_{\mathcal{V}} \sigma \cdot \mathsf{null}} \quad \text{(ad-addr)}\ \frac{\iota \in dom(h)}{h \vdash_{\mathcal{V}} \sigma \cdot \iota} \quad \text{(ad-new)}\ \frac{}{h \vdash_{\mathcal{V}} \sigma \cdot \mathsf{new}\, t} \quad \text{(ad-Var)}\ \frac{}{h \vdash_{\mathcal{V}} \sigma \cdot x} \quad \text{(ad-this)}\ \frac{}{h \vdash_{\mathcal{V}} \sigma \cdot \mathsf{this}} \quad \text{(ad-verEx)}\ \frac{}{h \vdash_{\mathcal{V}} F[\mathsf{verfExc}]}$$

$$\text{(ad-ass)}\ \frac{\begin{array}{l} h, \sigma \vdash e_r : \mathbb{r}\, c' \\ \mathcal{F}(c', f) = \_, c \\ \mathbb{r} \sqsubseteq \mathbb{U}_{\sigma,c} \\ h \vdash_{\mathcal{V}} \sigma \cdot e_r \\ h \vdash_{\mathcal{V}} \sigma \cdot e'_r \end{array}}{h \vdash_{\mathcal{V}} \sigma \cdot e_r.f = e'_r} \quad \text{(ad-fld)}\ \frac{h \vdash_{\mathcal{V}} \sigma \cdot e_r}{h \vdash_{\mathcal{V}} \sigma \cdot e_r.f} \quad \text{(ad-end)}\ \frac{h \vdash_{\mathcal{V}} \sigma' \cdot v}{h \vdash_{\mathcal{V}} \sigma \cdot \sigma' \cdot \mathsf{ret}\, v}$$

$$\text{(ad-call)}\ \frac{\begin{array}{l} h, \sigma \vdash e_r : \mathbb{r}\, c' \\ \mathcal{B}(c', m) = \_, c \\ \mathbb{r} \sqsubseteq \mathbb{C}_{\sigma,c,m} \\ h \vdash_{\mathcal{V}} \sigma \cdot e_r \\ h \vdash_{\mathcal{V}} \sigma \cdot e'_r \end{array}}{h \vdash_{\mathcal{V}} \sigma \cdot e_r.m(e'_r\ \mathsf{prv}\, \mathbb{B}_{\sigma,\mathbb{r}})} \quad \text{(ad-call-2)}\ \frac{\begin{array}{l} h, \sigma \vdash v : \mathbb{r}\, c' \\ \mathcal{B}(c', m) = \_, c \\ h \models \mathbb{B}_{\sigma,\mathbb{r}}, \sigma \\ \mathbb{r} \sqsubseteq \mathbb{C}_{\sigma,c,m} \\ h \vdash_{\mathcal{V}} \sigma \cdot v \\ h \vdash_{\mathcal{V}} \sigma \cdot v' \end{array}}{h \vdash_{\mathcal{V}} \sigma \cdot v.m(v')}$$

$$\text{(ad-start)}\ \frac{h \vdash_{\mathcal{V}} \sigma' \cdot e}{h \vdash_{\mathcal{V}} \sigma \cdot \sigma' \cdot \mathsf{call}\, e\ \mathsf{prv}\, \mathbb{E}_{\sigma'}} \quad \text{(ad-frame)}\ \frac{h \vdash_{\mathcal{V}} \sigma' \cdot e_r}{h \vdash_{\mathcal{V}} \sigma \cdot \sigma' \cdot \mathsf{ret}\, e_r\ \mathsf{prv}\, \mathbb{E}_{\sigma'}}$$

Fig. 9. Well-verified runtime expressions.

29. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM Press, 2005.
30. M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *POPL*, pages 75–86. ACM Press, 2008.
31. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, 1997.
32. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
33. J. P. Talpin and P. Jouvelot. The Type and Effect Discipline. In *LICS*, pages 162–173. IEEE Computer Society, 1992.

A Appendix—The Framework

Definition 9 *A runtime structure is a tuple*

$$\textsc{RStruct} = (\textsc{Hp}, \textsc{Adr}, \simeq, \preceq, dom, cls, fld, upd, new)$$

where $\textsc{Hp}$*, and* $\textsc{Adr}$ *are sets, and where*

$$\begin{array}{lll} \simeq\ \subseteq \textsc{Hp} \times \textsc{Hp} & \preceq\ \subseteq \textsc{Hp} \times \textsc{Hp} & dom : \textsc{Hp} \to \mathcal{P}(\textsc{Adr}) \\ cls : \textsc{Hp} \times \textsc{Adr} \rightharpoonup \textsc{Cls} & & new : \textsc{Hp} \times \textsc{Adr} \times \textsc{Typ} \to \textsc{Hp} \times \textsc{Adr} \\ fld : \textsc{Hp} \times \textsc{Adr} \times \textsc{Fld} \rightharpoonup \textsc{Val} & & upd : \textsc{Hp} \times \textsc{Adr} \times \textsc{Fld} \times \textsc{Val} \to \textsc{Hp} \end{array}$$

where $\textsc{Val} = \textsc{Adr} \cup \{\mathit{null}\}$ *for some element* $\mathit{null} \notin \textsc{Adr}$*. For all* $h \in \textsc{Hp}$*,* $\iota, \iota' \in \textsc{Adr}$*,* $v \in \textsc{Val}$*, we require:*

(H1) $\iota \in dom(h) \Rightarrow \exists c.cls(h, \iota) = c$

(H2) $h \simeq h' \Rightarrow dom(h) = dom(h'),\ \ cls(h, \iota) = cls(h', \iota)$

(H3) $h \preceq h' \Rightarrow dom(h) \subseteq dom(h'),\ \ \forall \iota \in dom(h).cls(h, \iota) = cls(h', \iota)$

(H4) $upd(h, \iota, f, v) = h' \Rightarrow \begin{cases} h \simeq h' \quad fld(h', \iota, f) = v, \\ \iota \neq \iota' \text{ or } f \neq f' \Rightarrow fld(h', \iota', f') = fld(h, \iota', f') \end{cases}$

(H5) $new(h, \iota, t) = h', \iota' \Rightarrow h \preceq h',\ \ \iota' \in dom(h') \backslash dom(h)$

Definition 10 *$E[\cdot]$ and $F[\cdot]$ are defined as follows:*

$$\begin{array}{l} E[\cdot] ::= [\cdot] \mid E[\cdot].f \mid E[\cdot].f = e \mid \iota.f = E[\cdot] \mid E[\cdot].m(e) \mid \iota.m(E[\cdot]) \mid E[\cdot]\,\textsf{\textit{prv}}\,\mathbb{p} \mid \textsf{\textit{ret}}\,E[\cdot] \\ F[\cdot] ::= [\cdot] \mid F[\cdot].f \mid F[\cdot].f = e \mid \iota.f = F[\cdot] \mid F[\cdot].m(e) \mid \iota.m(F[\cdot]) \mid F[\cdot]\,\textsf{\textit{prv}}\,\mathbb{p} \mid \sigma\cdot F[\cdot] \\ \qquad \mid \textsf{\textit{call}}\,F[\cdot] \mid \textsf{\textit{ret}}\,F[\cdot] \end{array}$$

Definition 11 *A programming language is a tuple*

$$PL \;=\; (\textsc{Prg}, \textsc{RStruct}, \mathrm{R}, \mathrm{P})$$

where R *and* P *are sets, and* Prg *is a set where every $\Pi \in$ Prg is a tuple*

$$\Pi \;=\; \begin{pmatrix} \mathcal{F}, \mathcal{M}, \mathcal{B}, <: \text{(class definitions)} & \sqsubseteq, [\![\cdot]\!] \text{ (inclusion and interpretations)} \\ \models, \vdash \quad \text{(invariant and type satisfaction)} & \end{pmatrix}$$

with signatures:

$$\begin{array}{llll} \mathcal{F} & : \textsc{Cls} \times \textsc{Fld} \rightharpoonup \textsc{Typ} \times \textsc{Cls} & \mathcal{M} & : \textsc{Cls} \times \textsc{Mthd} \rightharpoonup \textsc{Typ} \times \textsc{Typ} \\ \mathcal{B} & : \textsc{Cls} \times \textsc{Mthd} \rightharpoonup \textsc{Expr} \times \textsc{Cls} & & \\ <: & \subseteq \;\textsc{Cls} \times \textsc{Cls} \;\cup\; \textsc{Typ} \times \textsc{Typ} & \sqsubseteq & \subseteq \;\mathrm{R} \times \mathrm{R} \\ [\![\cdot]\!] & : \mathrm{R} \times \textsc{Hp} \times \textsc{Adr} \to \mathcal{P}(\textsc{Adr}) & [\![\cdot]\!] & : \mathrm{P} \times \textsc{Hp} \times \textsc{Adr} \to \mathcal{P}(\textsc{Adr} \times \textsc{Cls}) \\ \models & \subseteq \;\textsc{Hp} \times \textsc{Adr} \times \textsc{Cls} & \vdash & \subseteq \;(\textsc{Env} \times \textsc{Expr} \cup \textsc{Hp} \times \textsc{RExpr}) \times \textsc{Typ} \end{array}$$

where every $\Pi \in$ Prg must satisfy the constraints:

(P1) $\mathcal{F}(c, f) = t, c' \Rightarrow c <: c'$ *(P2)* $\mathcal{B}(c, m) = e, c' \Rightarrow c <: c'$
(P3) $\mathcal{F}(\mathit{cls}(h, \iota), f) = t, \_ \Rightarrow \exists v.\mathit{fld}(h, \iota, f) = v$ *(P4)* $\mathbb{r}_1 \sqsubseteq \mathbb{r}_2 \Rightarrow [\![\mathbb{r}_1]\!]_{h,\iota} \subseteq [\![\mathbb{r}_2]\!]_{h,\iota}$
(P5) $[\![\mathbb{r}]\!]_{h,\iota} \subseteq \mathit{dom}(h)$ *(P6)* $h \preceq h' \Rightarrow [\![\mathbb{p}]\!]_{h,\iota} \subseteq [\![\mathbb{p}]\!]_{h',\iota}$
(P7) $\mathbb{r}\,c <: \mathbb{r}'\,c' \Rightarrow \mathbb{r} \sqsubseteq \mathbb{r}',\ c <: c'$

Definition 12 *Stack $\overline{\sigma}$ is* valid *w.r.t. heap h in a verification technique $\mathcal{V}$, denoted by $h \vdash_{\mathcal{V}} \overline{\sigma}$, iff:*

$$\overline{\sigma} = \overline{\sigma_1}\cdot\sigma\cdot\sigma'\cdot\overline{\sigma_2} \Rightarrow \sigma' = (\iota, \_, c', m), \quad h, \sigma \vdash \iota : \mathbb{r}\,\_, \quad c' <: c, \quad \mathbb{r} \sqsubseteq \mathbb{C}_{\sigma,c,m}$$

Definition 13 *The function stack* : RExpr $\to$ Stk* *yields the stack of a runtime expression:*

$$\mathit{stack}(E[\mathrm{e}_r]) = \begin{cases} \sigma\cdot\mathit{stack}(\mathrm{e}'_r) & \text{if } \mathrm{e}_r = \sigma\cdot\mathrm{e}'_r \\ \epsilon & \text{otherwise} \end{cases}$$

Definition 14 *For every program, the judgement:*

$$\vdash_{\!\mathit{wf}} \;:\; (\textsc{Hp} \times \textsc{Stk} \times \textsc{Stk} \times \mathrm{R}) \;\cup\; (\textsc{Env} \times \textsc{Hp} \times \textsc{Stk}) \;\cup\; \textsc{Prg} \quad \text{is defined as:}$$

- $\vdash_{\!\mathit{wf}} \Pi \;\Leftrightarrow\; \begin{cases} \text{(F1)} & \mathcal{M}(c, m) = t, t' \;\Rightarrow\; \exists e.\ \mathcal{B}(c, m) = e, \_, \quad c, m, t \vdash e : t' \\ \text{(F2)} & c <: c',\ \mathcal{F}(c', f) = t, c'' \;\Rightarrow\; \mathcal{F}(c, f) = t', c'',\ t' = t \\ \text{(F3)} & c <: c',\ \mathcal{M}(c, m) = t, t',\ \mathcal{M}(c', m) = t'', t''' \;\Rightarrow\; t = t'',\ t' = t'''' \\ \text{(F4)} & c <: c',\ \mathcal{B}(c', m) = e', c'' \;\Rightarrow\; \exists c'''.\ \mathcal{B}(c, m) = e, c''',\ c''' <: c'' \end{cases}$
- $h, \sigma \vdash_{\!\mathit{wf}} \sigma' : \mathbb{r} \;\Leftrightarrow\; \sigma' = (\iota, \_, \_, \_), \quad h, \sigma \vdash \iota : \mathbb{r}\,\_$
- $\Gamma \vdash_{\!\mathit{wf}} h, \sigma \;\Leftrightarrow\; \begin{cases} \exists c, m, t, \iota, v. \quad \Gamma = c, m, t, \quad \sigma = (\iota, v, c, m), \\ \qquad\qquad \mathit{cls}(h, \iota) <: c, \quad h, \sigma \vdash v : t \end{cases}$

Definition 15 *A programming language* PL *has a* sound type system *if all programs $\Pi \in$ PL satisfy the constraints:*

(T1) $\Gamma \vdash e : t,\ t <: t' \;\Rightarrow\; \Gamma \vdash e : t'$ *(T2)* $h \vdash \mathrm{e}_r : t,\ t <: t' \;\Rightarrow\; h \vdash \mathrm{e}_r : t'$
(T3) $h \vdash \mathrm{e}_r : t,\ h \simeq h' \;\Rightarrow\; h' \vdash \mathrm{e}_r : t$ *(T4)* $h \vdash \sigma\cdot\iota : \_\,c \;\Rightarrow\; \mathit{cls}(h, \iota) <: c$
(T5) $h \vdash \sigma\cdot\iota.m(v) : t \;\Rightarrow\; h \vdash \sigma\cdot\iota : \mathbb{r}\,c\ \mathcal{M}(c, m) = t', t,\ h \vdash \sigma\cdot v : t'$
(T6) $\sigma = (\iota, \_, \_, \_),\ h \vdash \sigma\cdot\iota' : \mathbb{r}\,\_ \;\Rightarrow\; \iota' \in [\![\mathbb{r}]\!]_{h,\iota}$
(T7) $\Gamma \vdash e : \mathbb{r}\,c,\ \Gamma \vdash h, \sigma \;\Rightarrow\; h \vdash \sigma\cdot e : \mathbb{r}\,c$
(T8) $\vdash_{\!\mathit{wf}} \Pi,\ \ h, \sigma \vdash \mathrm{e}_r : t\ \ \mathrm{e}_r, h \longrightarrow \mathrm{e}'_r, h' \;\Rightarrow\; h', \sigma \vdash \mathrm{e}'_r : t$

B Appendix—The Instantiations

Müller et al. We assume an additional heap operation, which gives an object's owner: $own\ :\ \text{HP} \times \text{ADR} \rightarrow \text{ADR} \times \text{CLS}$.
Regions are interpreted as follows:

$$\begin{aligned}
\llbracket \mathsf{self} \rrbracket_{h,\iota} &= \{\iota\} & \llbracket \mathsf{any} \rrbracket_{h,\iota} &= dom(h) \\
\llbracket \mathsf{rep}\langle c\rangle \rrbracket_{h,\iota} &= \big\{\iota' \mid\ own(h,\iota') = \iota\, c\big\} & \llbracket \mathsf{emp} \rrbracket_{h,\iota} &= \emptyset \\
\llbracket \mathsf{peer} \rrbracket_{h,\iota} &= \big\{\iota' \mid\ own(h,\iota') = own(h,\iota)\big\} & \llbracket \mathbb{r}_1 \sqcup \mathbb{r}_2 \rrbracket_{h,\iota} &= \llbracket \mathbb{r}_2 \rrbracket_{h,\iota} \cup \llbracket \mathbb{r}_2 \rrbracket_{h,\iota}
\end{aligned}$$

Properties are interpreted as follows:

$$\begin{aligned}
\llbracket \mathsf{self}\langle c\rangle \rrbracket_{h,\iota} &= \{(\iota,c) \mid\ cls(h,\iota) <: c\} & \llbracket \mathsf{emp} \rrbracket_{h,\iota} &= \emptyset \\
\llbracket \mathsf{peer}\langle c\rangle \rrbracket_{h,\iota} &= \{(\iota',c') \mid own(h,\iota') = own(h,\iota) \wedge \mathsf{vis}(c',c)\} \\
\llbracket \mathsf{rep} \rrbracket_{h,\iota} &= \{(\iota',c') \mid own(h,\iota') = \iota\ \_\} & \llbracket \mathsf{rep}^+ \rrbracket_{h,\iota} &= \llbracket \mathsf{rep} \rrbracket_{h,\iota} \cup \llbracket \mathsf{rep};\mathsf{rep}^+ \rrbracket_{h,\iota} \\
\llbracket \mathsf{own} \rrbracket_{h,\iota} &= \{own(h,\iota)\} & \llbracket \mathsf{own}^+ \rrbracket_{h,\iota} &= \llbracket \mathsf{own} \rrbracket_{h,\iota} \cup \llbracket \mathsf{own};\mathsf{own}^+ \rrbracket_{h,\iota} \\
\llbracket \mathsf{super}\langle c\rangle \rrbracket_{h,\iota} &= \{(\iota,c') \mid\ c <: c'\} & \llbracket \mathbb{p}_1;\mathbb{p}_2 \rrbracket_{h,\iota} &= \textstyle\bigcup_{(\iota',c)\in\llbracket \mathbb{p}_1 \rrbracket_{h,\iota}} \llbracket \mathbb{p}_2 \rrbracket_{h,\iota'}
\end{aligned}$$

The predicate $\mathsf{intrsPeer}(\mathbb{r})$, is defined as:

$$\begin{aligned}
&\mathsf{intrsPeer}(\mathsf{emp}) = \mathsf{intrsPeer}(\mathsf{rep}\langle c\rangle) = \mathit{false} \\
&\mathsf{intrsPeer}(\mathsf{self}) = \mathsf{intrsPeer}(\mathsf{peer}) = \mathsf{intrsPeer}(\mathsf{any}) = \mathit{true} \\
&\mathsf{intrsPeer}(\mathbb{r}_1 \sqcup \mathbb{r}_2) = \mathsf{intrsPeer}(\mathbb{r}_1)\ ||\ \mathsf{intrsPeer}(\mathbb{r}_2)
\end{aligned}$$

Lu et al. We interpret regions as follows:

$$\begin{aligned}
&\llbracket \mathsf{emp} \rrbracket_{h,\iota} = \emptyset \qquad \llbracket \mathsf{self} \rrbracket_{h,\iota} = \{\iota\} \qquad \llbracket \mathbb{r} \sqcup \mathbb{r}' \rrbracket_{h,\iota} = \llbracket \mathbb{r} \rrbracket_{h,\iota} \cup \llbracket \mathbb{r}' \rrbracket_{h,\iota} \\
&\llbracket c\langle \overline{K}\rangle \rrbracket_{h,\iota} = \big\{\iota' \mid h \vdash \iota' : c\langle\bar{\iota}\rangle, \forall i.\, \iota_i \in \{\!|K_i|\!\}_{h,\iota}\big\}
\end{aligned}$$

As usual in ownership systems, $h \vdash \iota : c\langle\bar{\iota}\rangle$ describes that ι points to an object of a subclass of $c\langle\bar{\iota}\rangle$, while $h \vdash \iota' \preceq \iota$ expresses that ι' is owned by ι, and $h \vdash \iota' \preceq^* \iota$ is the transitive closure. We interpret properties as follows:

$$\begin{aligned}
\llbracket \mathsf{emp} \rrbracket_{h,\iota} &= \llbracket \mathsf{top} \rrbracket_{h,\iota} = \llbracket \mathsf{bot} \rrbracket_{h,\iota} = \emptyset \quad \llbracket \mathsf{self} \rrbracket_{h,\iota} = \{(\iota,c) \mid ...\} \\
\llbracket K \rrbracket_{h,\iota} &= \big\{(\iota',c) \mid \iota' \in \{\!|K|\!\}_{h.\iota},\ cls(h,\iota') <: c\big\} \\
\llbracket K;\mathbb{p} \rrbracket_{h,\iota} &= \begin{cases} all(h) & K = \mathsf{top}, \mathbb{p} = \mathsf{rep}^* \ \vee K = \mathsf{bot}, \mathbb{p} = \mathsf{own}^* \\ \bigcup_{(\iota',c)\in\llbracket K \rrbracket_{h,\iota}} \llbracket \mathbb{p} \rrbracket_{h,\iota'} & \mathbb{p} \in \{\mathsf{rep}^*, \mathsf{own}^*\} \end{cases} \\
\llbracket \mathsf{rep}^* \rrbracket_{h,\iota} &= \big\{\iota' \mid h \vdash \iota' \preceq^* \iota\big\} \qquad \llbracket \mathsf{own}^* \rrbracket_{h,\iota} = \big\{\iota' \mid h \vdash \iota \preceq^* \iota'\big\} \\
\{\!|X|\!\}_{h,\iota} &= \big\{\iota_i \mid h \vdash \iota : c\langle\bar{\iota}\rangle,\ c \text{ has formal parameters } \bar{X},\ X = X_i\big\}
\end{aligned}$$

The owner extraction function $\mathcal{O}$ is defined as:

$$\mathcal{O}_{\mathbb{r},c} = \begin{cases} K_1, & \text{if } \ \mathbb{r} = c\langle\overline{K}\rangle \\ X_1, & \text{if } \ \mathbb{r} = \mathsf{self}, \text{ class } c \text{ has formal parameters } \bar{X}. \\ \bot & \text{otherwise} \end{cases}$$

A Universe-Type-Based Verification Technique for Mutable Static Fields and Methods – Work in progress –

A. J. Summers$^{(1)}$, S. Drossopoulou$^{(1)}$, and P. Müller$^{(2)}$
$^{(1)}$ Imperial College London, $^{(2)}$ Microsoft Research, Redmond

Abstract. We present a novel technique for the verification of invariants in the setting of a Java-like language including static fields and methods. The technique is a generalisation of the existing Visibility Technique of Müller et al., which employs universe types.
In order to cater for mutable static fields, we extend this topology to multiple trees (a forest), where each tree is rooted in a class. This allows classes to naturally own object instances as their static fields. We describe how to extend the Visibility Technique to this topology, incorporating extra flexibility for the treatment of static methods.
We encounter a potential source of callbacks not present in the original technique, and show how to overcome this using an effects system. To allow flexible and modular verification, we refine our topology with a hierarchy of 'levels'.

1 Introduction

In this paper, we extend the Visibility Technique (VT for short) [10], a known visible states verification technique based on universe types, to cater for static fields and methods. When adding statics to verification, one needs to address the following questions:

1. Where in the topology do static fields appear?
2. May instance methods update static fields?
3. May static invariants mention the fields of objects of their class?
4. May instance invariants mention static fields of their class, or of other classes?
5. Can static methods break invariants of objects, and if so, of which objects?
6. Can instance methods break static invariants, and if so, of which classes?
7. What proof obligations are necessary before a call to a static method?
8. What proof obligations are necessary before a call to an instance method?

In this paper, we explore these questions in the context of VT, and extend the technique and heap topology to handle static fields. In the process, we encounter a potential source of callbacks not present in VT, and devote much of this paper to solving this problem. We develop an approach involving a combination of effect annotations and refinements to the heap topology using levels. We then extend the technique to allow more expressive invariants.

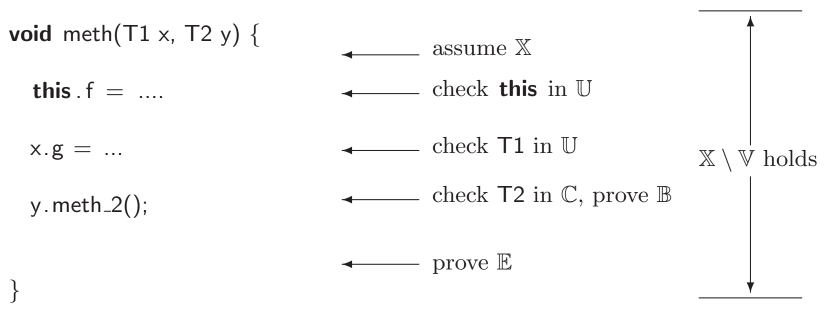


Fig. 1. Illustration of the use of the seven components.

In Sec. 2 we give the background to visible states verification techniques, universe types, and VT. In Sec. 3 we discuss the first two questions from above. In Sec. 4 we address the others, give a first attempt to an extension of VT, and argue that it is sound. We refine our approach with improved calculations of effects in Sec. 5, and with more powerful static class invariants in Sec. 6. In Sec. 7 we conclude. Proof sketches can be found in the longer version of our work, at `http://www.doc.ic.ac.uk/~ajs300m/papers/staticsFull.pdf`.

2 Background

Visible state verification techniques are defined around the notion of *visible states*, which correspond to the beginning and the end of any method call. At these visible states, the invariants of certain objects (exactly *which* objects depends on the contents of the call stack, and on the particular technique) are guaranteed to hold.

Several visible states techniques have been suggested, *e.g.*, [12, 3, 10, 8], and they share many commonalities. As suggested in [2], these commonalities, as well as the differences, can be neatly distilled in terms of the following seven components:

$\mathbb{X}$ invariants e<u>x</u>pected to hold in visible states.
$\mathbb{V}$ invariants *<u>v</u>ulnerable* to a method, *i.e.*, which may be broken while it executes.
$\mathbb{D}$ invariants that may <u>d</u>epend on a given heap location[1].
$\mathbb{B}$ invariants that must be proven to hold <u>b</u>efore a method call.
$\mathbb{E}$ invariants that must be proven to hold at the <u>e</u>nd of a method body.
$\mathbb{U}$ permitted receivers for field <u>u</u>pdates.
$\mathbb{C}$ permitted receivers for method <u>c</u>alls.

The use of these components should be clear from their description above, but is also shown in Fig. 1 through annotating a method `meth1`: $\mathbb{X}$ may be assumed to hold in the pre- and post-states of the method. Between these visible states, some object invariants may be broken, but $\mathbb{X} \setminus \mathbb{V}$ is guaranteed to hold. Field updates and method calls are allowed if the receiver object is in $\mathbb{U}$ and $\mathbb{C}$, respectively. Before a method call, $\mathbb{B}$ must be proven. At the end of the method body, $\mathbb{E}$ must be proven. Finally, assignments to **this**.f and x.g affect at most $\mathbb{D}$.

[1] This also characterises indirectly the locations an invariant may depend on.

In [2], five *soundness conditions* are presented, and it is proven that if these conditions are satisfied, then the technique is sound (the expected invariants hold at visible states). In this paper, we use the framework of [2] informally, since the technique presented here does not quite fit the present formalism. However, the soundness conditions still guided us in the design of our technique. Informally, the five sufficient soundness conditions can be described as follows:

Definition 1 (Soundness Conditions).

1. $\mathbb{X}_{m'} \setminus (\mathbb{X}_m \setminus \mathbb{V}_m) \subseteq \mathbb{B}$
   When a legal (according to the technique, i.e., $\mathbb{C}$*) call is made to a method* m' *from a method* m*, all of the invariants which are both expected to hold by the new method* ($\mathbb{X}_{m'}$)*, and are not currently known to hold in the calling method (*i.e.*, not within* $\mathbb{X}_m \setminus \mathbb{V}_m$*), must be within the proof obligations made before the method call* ($\mathbb{B}$)*.*
2. $\mathbb{V} \cap \mathbb{X} \subseteq \mathbb{E}$
   The invariants both expected ($\mathbb{X}$) *by and vulnerable to* ($\mathbb{V}$) *a method, must be within the proof obligations at the end of the method* ($\mathbb{E}$)*.*
3. $\mathbb{V}_{m'} \setminus \mathbb{E}_{m'} \subseteq \mathbb{V}_m$
   If a (legal) method call is made to a method m' *from a method* m*, any invariants which are vulnerable to* m' *and not reestablished by* m'*, must be vulnerable to* m*.*
4. $\mathbb{D} \subseteq \mathbb{V}$
   Invariants depending on fields which may be legally modified (according to the technique, i.e., $\mathbb{U}$*) by a method, are vulnerable to the method.*
5. $\mathbb{X}_{c'} \subseteq \mathbb{X}_c$ *and* $(\mathbb{V}_{c'} \setminus \mathbb{E}_{c'}) \subseteq (\mathbb{V}_c \setminus \mathbb{E}_c)$
   If a method is overridden, then in the subclass version, no more invariants may be expected or left broken than in the superclass version.

One such visible states technique, the Visibility Technique (VT), was developed on top of universe types [10] with the aim to guide the verification process, and to guarantee modularity. Universe types [9] organise the heap into a tree topology, in which each object is *owned* by another object, and where an object o considers another object o', as its peer if they have the same direct owner; it considers it its rep if it is its direct owner[2]. The *owner-as-modifier discipline* (hereafter OAM) restricts field updates and method calls, implying in particular that the receivers of methods are only allowed to be reps or peers. Thus, at any time in execution any receiver on the call stack[3] is directly followed either by a rep or a peer. In Fig. 2, note that calls may only go "down" or "sideways".

The seven components from before have the following meaning for VT (we simplify slightly with respect to visibility, and to the exact class whose invariant we are considering):

[2] We do not discuss any or readonly references, nor pure methods.

[3] consisting of a sequence of activation records, each of which contains the then-current receiver

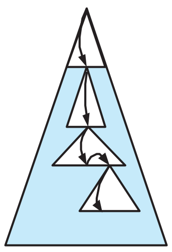


Fig. 2. Ownership Tree and Control Flow; the arrows show consecutive method calls and their receivers; note that calls go only "down", *i.e.*, to reps, or "sideways", *i.e.*, to peers. The shaded area indicates the area where objects satisfy their invariants.

$\mathbb{X}$ invariants of objects (reflexively, transitively) owned by peers.
$\mathbb{V}$ invariants of all transitive owners of the current receiver, plus invariants of peers of the current receiver.
$\mathbb{D}$ Invariants of peers and transitive owners may depend on the fields of an object.
$\mathbb{B}$ If the callee is a peer of the current receiver, then the invariants of all peers must be established. Otherwise, no proof obligations.
$\mathbb{E}$ the invariants of all visible peers.
$\mathbb{U}$ A field of an object may only be assigned to by the object's owner, or by any of its peers.
$\mathbb{C}$ A call is allowed if the callee is a peer or rep of the current receiver.

It can be shown that these parameters satisfy the soundness conditions of Def. 1 [2]. In particular, $\mathbb{X}$ and $\mathbb{V}$ and the owner-as-modifier discipline, guarantee that at any given time in execution, all objects are valid, except for those directly owned by one of the receivers on the call stack, *cf.* Fig. 2.

3 Heap Topology for Static Fields

The fundamental premise of this work is that classes should be able to own objects in the same way that other objects can. For example, if the behaviour of a class depends on a static field (to manage object creation, etc.) then this static field naturally 'belongs' to the inner workings of the class: its representation. This gives a natural interpretation of static rep fields: they should be treated analogously to instance rep fields, but with a class as their owner [7].

Thus, we extend our heap topology to include classes. Classes are the 'roots' of trees in our topology. As there are generally several classes in a program, our topology should allow for several such trees; we work with a *forest*. Furthermore, with classes acting as roots, there is no longer a need for an abstract **root** entity; these class-rooted trees make up the entire picture. Note that there are no objects at the 'same level' as the class entities, and classes do not have owners. In this paper, we do not consider a notion of static peer fields.

We interpret static fields and methods as instance fields and methods of the corresponding class object. That is, the class object (or class for short) is the receiver for an execution of a static method. We expect that modifications to static fields will be achieved by calling a static method of the class that declares the field. In other words, static methods may update the fields of their receiver class, just like instance methods in VT may update fields of their receiver object.

To summarise the ideas so far:

1. Each point in our heap topology corresponds to either an object or a class.
2. Objects (but not classes) each have exactly one owner (a class or an object).
3. The current receiver (on the stack) can be either an object or a class.

4 Basic Technique

Having defined a suitable heap topology, in this section we generalise VT to our setting.

A key aspect of our technique is that we preserve the OAM property of VT. In the following technique, control is only allowed to enter a tree in the heap topology via the 'root'; *i.e.*, by calling a static method on the class at the root of the tree. Instance method calls are restricted in the same way as in VT. This implies the following property, which will be useful for our reasoning:

Proposition 1. *A call stack (including the current method-call) always starts with a class receiver. If an object o is a receiver on the call stack, then the most recently-preceding class receiver on the call stack is the owner of the tree in which o resides.*

For the moment, we treat static invariants analogously to VT instance invariants. Therefore, they can only mention expressions which start with the static fields of the same class (since they have no peers).

How then, to handle static method calls? According to VT, a method call is only allowed if the current receiver is either the owner or a peer of the callee receiver. Since classes do not have either owners or peers, this would make static methods impossible to call. We initially considered allowing arbitrary static method calls. This immediately creates problems with callbacks; in particular, how do we know the invariants of the new receiver hold when we make the call? If our current call stack has already visited this class, we may have left invariants broken.

We solve this problem by the following rule: a static method may only be called on a class c, if c has not been a previous receiver on the call stack. However, this rule is slightly too restrictive, since it unnecessarily prohibits a static method of class c from calling another static method of class c. Our rule of thumb is:

> *A static method of c can be called if either c is the current receiver, or c is not already a receiver on the call stack.*

We are now in a position to define our technique in terms of the seven components. Compared with the description of VT, we need to extend $\mathbb{X}$ to reflect which invariants in other trees are expected, depending on the current call stack, and $\mathbb{C}$ to reflect the special rules for static method calls. The other five parameters are straightforward generalisations of those for VT. We highlight the differences between our work and VT in italics, and point out the interpretation of these components with regard to a static method call in footnotes.

$\mathbb{X}$ invariants of objects (reflexively, transitively) owned by peers, *plus all invariants in trees not currently visited on the call stack*[4].
$\mathbb{V}$ invariants of all transitive owners of the current receiver, plus invariants of peers of the current receiver[5].
$\mathbb{D}$ Invariants of peers and transitive owners may depend on the field of an object *or class*[6].
$\mathbb{B}$ If the callee is a peer of the current receiver, then the invariants of all peers must be established. Otherwise, no proof obligations[7].
$\mathbb{E}$ the invariants of all visible peers[8].
$\mathbb{U}$ A field of an object *or class* may only be assigned to by its owner, or by any of its peers[9].
$\mathbb{C}$ A call to an instance method is allowed if the callee is a peer or rep of the current receiver. *A call to a static method m on class c is allowed if either the current receiver is c itself, or else c is not on the current call stack.*

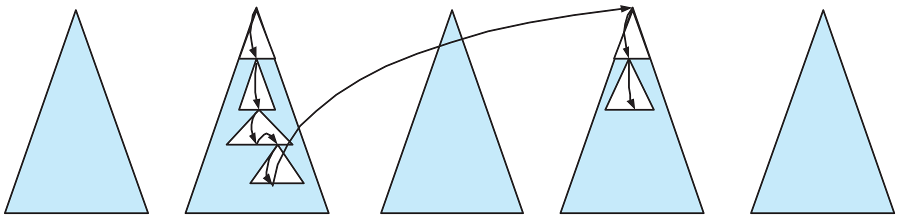


Fig. 3. Calls stacks across several trees, invariants hold in shaded areas.

When considering only the tree of the current receiver, the rules are essentially those of VT. However, the other trees either have none of their invariants expected, or all of them, depending on whether or not they have been visited on the current call stack. Furthermore, static methods are treated differently from

[4] For a static method, this amounts to all the invariants of the current tree, plus each unvisited tree.
[5] The only invariants vulnerable to a call of a static method in class c are the static invariants of c itself.
[6] The only invariants which are allowed to depend on a static field declared in class c are the static invariants of c.
[7] If a static method is called on a class c which is both caller and callee (a 'self' call), then the static invariants of c must be reestablished first.
[8] For a static method, the invariants of the class.
[9] A static field can only be assigned to by the class itself.

instance method calls, in that any call is permitted so long as the callee has not been a receiver prior to the current one on the call stack.

Since $\mathbb{C}$ depends on the current call stack, it is not possible to statically verify whether a method call will be legal. We therefore identify next a way of conservatively approximating when method calls are legal.

Effect Annotations. For each class c and method m, we require a set of *effects*, $\mathcal{E}ffs(c,m)$, predicting which classes may have static methods called on them as a result of calling m of c. $\mathcal{E}ffs(c,m)$ is a (possibly empty) set of class names. This is described by requirements 1-3 in Def. 2 below.

If, from within the body of a static method m of class c, we make a call to a (static or instance) method m' defined in class c' (with a different receiver), and if this method call may eventually result in a callback to c, then as a consequence of Def. 2, we must have $c \in \mathcal{E}ffs(c',m')$. Therefore, we can rule out dangerous callbacks on c by insisting that any method which is called from a static method of c does not contain c in its effects. This is described through the method restriction in item 4 of Def. 2.

Definition 2 (Valid Effects and Method Restrictions).

1. *Within the body of a method m of class c, if there is a call $e.m'(\ldots)$ and e has static type c', then $\mathcal{E}ffs(c',m') \subseteq \mathcal{E}ffs(c,m)$.*
2. *Within the body of a method m of class c, if there is a call $c'.m'(\ldots)$ to a static method m' of class c', then*
   (a) $\mathcal{E}ffs(c',m') \subseteq \mathcal{E}ffs(c,m)$ and
   (b) if m is an instance method or $c \neq c'$ [10], then $c' \in \mathcal{E}ffs(c,m)$.
3. *If c' is a subclass of c which overrides a method m, then $\mathcal{E}ffs(c',m) \subseteq \mathcal{E}ffs(c,m)$.*
4. *A static method m of c is legal, only if $c \notin \mathcal{E}ffs(c,m)$.*

Soundness. We focus on the first item from Def. 1: the guarantee that when a method call is made, the invariants expected in the new method will hold (because they have been preserved, or proven before the call is made). We claim that the other points can be easily established.

In the technique presented, all invariants may only depend on the fields of peers (if any) and any objects transitively owned. Furthermore, fields may only be modified by peers. Therefore, we have the following property:

Proposition 2 (Broken Invariants). *If, at runtime, the invariants of an object (or class) do not hold, then one of the receivers on the call stack (possibly the current one) must be the object (or class) itself or one of its peers.*

[10] *i.e.*, a static method always may call another static method from the same class.

To demonstrate that our restrictions using effects (Def. 2) are sufficient to guarantee that our desired notion of valid method call ($\mathbb{C}$) is always adhered to, we need a deeper discussion of possible sequences of calls. We require some notation to capture these sequences; we wish to track the receiver-method pairs from (consecutive) fragments of the call stack. We write (c, m) for a call of static method m on class c, and (o, m) for a call of instance method m on object o. For any receivers r,r' (which may each be either classes or objects), we write $(r, m)\ \overline{call}\ (r', m')$ to denote a sequence of legal calls[11] beginning with m and ending with m', *i.e.*, method m on receiver r calls some method m_1 on some receiver r_1, etc., which eventually leads to calling method m' on receiver r'. These sequences of calls correspond to consecutive regions of a call stack, in which only the receiver and method information is retained. Note that such sequences need not begin from the initial (class) receiver of a call stack. We consider only call-sequences which are legal according to our technique.

We can now show how calls are restricted by the effect annotations:

Proposition 3 (Effects are Conservative).

1. *For any call-sequence $(o, m)\ \overline{call}\ (c', m')$, if c is the dynamic class of o, then $c' \in \mathcal{E}ffs(c, m)$.*
2. *For any call-sequence $(c, m)\ \overline{call}\ (c', m')$, if either $c \neq c'$ or any of the intermediate receivers in $\overline{call}$ are not c, then $c' \in \mathcal{E}ffs(c, m)$.*
3. *Any call-sequence $(c, m)\ \overline{call}\ (c, m')$ consists only of calls where c is the receiver.*
4. *If o and o' are peers, then any call-chain $(o, m)\ \overline{call}\ (o', m')$ features only peers of o (and o') as receivers.*

Finally, we can prove that the invariants of a new receiver are always guaranteed by the proof obligations in the technique:

Theorem 1.

1. *If a static method m is to be called on c, then the proof obligations imposed by the technique guarantee that c's invariants hold.*
2. *If an instance method m is to be called on o, then the proof obligations imposed by the technique guarantee that o's invariants hold.*

5 Refined Effects

The effects as described so far require annotations for *all* classes used in a program. This requirement leads to a high annotation burden, compomises information hiding, and limits the usability of the technique presented so far, as the following example illustrates.

[11] *i.e.*, calls which are permissible according to Def. 2.

Example 1 (Method Overriding and Effects). Consider the String class of the Java API. An implementation of this class can exploit that fact that strings are immutable in Java, and so share instances of objects, by using static fields from class String to maintain a 'pool' of used String instances. This would imply that the constructor String calls String static methods, and would have String in its effects. Consider now that we want to write a class which overrides the equals() method inherited from Object:

```
class MyClass extends Object{
  boolean equals(Object o)
  {
    System.out. println (new String("equals()  called"));
    return this == o;
  }
}
```

Obviously, we need to have String$\in \mathcal{E}ffs$(MyClass,equals), and because of Def. 2 (item 3), we also need that String$\in \mathcal{E}ffs$(Object,equals). But, it is unlikely that this effect was predicted when the class Object was given effect annotations. Therefore, this method definition would be illegal. This illustrates an annotation problem (annotations may need recomputing), an information-hiding problem (our code should not need to know how String is implemented), and a usability problem (our technique forbids this method declaration).

To alleviate this burden, we introduce a refinement, whereby we group classes in a linear hierarchy of 'levels', such that the code of lower-level classes does not mention the higher-level classes[12]. The intuition is that library classes should have been previously verified and belong on a 'lower level' than the classes which the programmer is now writing. We express the levels through a function $\mathcal{L}vl(\_)$ which maps classes to integers.

Definition 3 (Valid Levels). *c mentions* $c' \Rightarrow \mathcal{L}vl(c) \geq \mathcal{L}vl(c')$.

Because classes in the lower levels do not 'know about' classes in the upper levels, it is impossible for them to make static calls on the classes in the upper levels (*cf.* Fig. 4). Therefore, if we consider verification of the topmost level, then when a call is made down to a lower level, the effect annotations are no longer necessary. [13] Thus, we refine our effect annotation sets to only mention classes on the same level as the method being verified. The new conditions on effects (in which differences in comparison with Def. 2 are shown in roman font) are:

Definition 4 (Refined Effects).

[12] For example, we could consider the Java API classes (*e.g.*, Object and String) to be on a lower level than our classes, and it would be naturally guaranteed that the API classes do not mention ours.

[13] To handle dynamic binding, we require the effects of methods that override methods in lower levels to be empty and, thus, independent of the effects of the overridden method.

1. If c' is in $\mathcal{E}\!f\!f\!s(c, m)$ then $\mathcal{L}vl(c') = \mathcal{L}vl(c)$.
2. Within the body of a method m of class c, if there is a call e.m′(...) and e has static type c′, and $\mathcal{L}\mathbf{vl}(\mathbf{c}) = \mathcal{L}\mathbf{vl}(\mathbf{c'})$, *then* $\mathcal{E}\!f\!f\!s(c', m') \subseteq \mathcal{E}\!f\!f\!s(c, m)$.
3. Within the body of a method m of class c, if there is a call c′.m′(...) to a static method m′ of class c′ and $\mathcal{L}\mathbf{vl}(\mathbf{c}) = \mathcal{L}\mathbf{vl}(\mathbf{c'})$, *then:*
   (a) $\mathcal{E}\!f\!f\!s(c', m') \subseteq \mathcal{E}\!f\!f\!s(c, m)$
   (b) if m is either an instance method or $c \neq c'$, *then* $c' \in \mathcal{E}\!f\!f\!s(c, m)$.
4. If c′ is a subclass of c which overrides a method m, then
   (a) If $\mathcal{L}\mathbf{vl}(\mathbf{c}) = \mathcal{L}\mathbf{vl}(\mathbf{c'})$, then $\mathcal{E}\!f\!f\!s(c', m) \subseteq \mathcal{E}\!f\!f\!s(c, m)$
   (b) If $\mathcal{L}\mathbf{vl}(\mathbf{c}) < \mathcal{L}\mathbf{vl}(\mathbf{c'})$, then $\mathcal{E}\mathbf{ffs}(\mathbf{c'}, \mathbf{m}) = \emptyset$
5. A static method m of c is legal, only if $c \notin \mathcal{E}\!f\!f\!s(c, m)$.

The refined conditions given permit smaller effects sets for methods than those of Def. 2. Considering the example at the start of the section, it is no longer necessary (or indeed, allowed) for String to be in $\mathcal{E}\!f\!f\!s$(MyClass,equals).

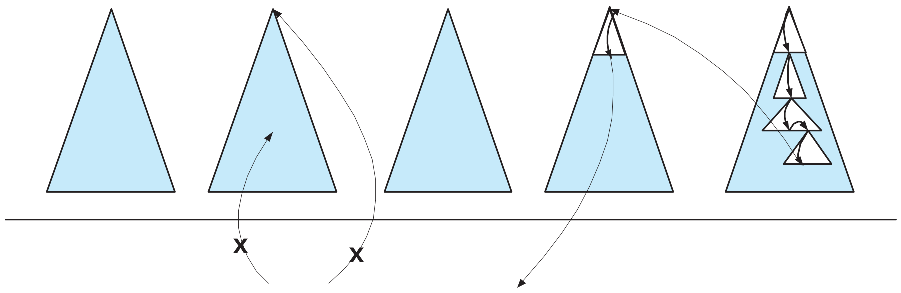


Fig. 4. Trees in one level. The current level may call into the lower level, but no calls from the lower level may come into the current level. The level of an object is determined by the class that transitively owns the object, not by the object's type.

Soundness. As in the previous section, we focus on ensuring that the proof obligations made before method calls are always sufficient to guarantee the expected invariants. Furthermore, we make the assumption here that we are only interested in verifying the 'top-level'; we assume that the classes on lower levels have already been verified. This can be used to construct an inductive verification of the entire class-structure, if needed, but also allows us a more-modular approach; once the classes on a lower level have been verified, we need not repeat the process if we are only adding classes to higher levels.

We write $\mathcal{L}vl(o)$ for the level of an object, defined to be the level of the class which transitively owns the object (*i.e.*, the class which is the 'root' of the appropriate tree). We can then show the following property:

Proposition 4 (Levels do not Increase through Calls).

1. *If object o is transitively owned by class c, and if c' is the dynamic class of o, then $\mathcal{L}vl(c) \geq \mathcal{L}vl(c')$.*
2. *For any call-sequence $(c, m)\ \overline{call}\ (o, m')$, where $\overline{call}$ consists exclusively of instance method calls, if c' is the dynamic class of o, then $\mathcal{L}vl(c) \geq \mathcal{L}vl(c')$.*
3. *For any sequence of calls $(r_1, m_1)\ \overline{call}\ (r_2, m_2)$, in which r_1, r_2 can be any receivers,* i.e.*, classes or objects, we have $\mathcal{L}vl(r_1) \geq \mathcal{L}vl(r_2)$.*
4. *For any call-sequence $(c, m)\ \overline{call}\ (c, m')$, for all the intermediate receivers r, we have $\mathcal{L}vl(r) = \mathcal{L}vl(c)$.*

This allows us to construct similar arguments to those in the previous section, regarding soundness of method calls. Proposition 2 still holds for this refinement. Proposition 3 holds in the restricted case that all receivers involved are from the top-level. Theorem 1 then holds for all such receivers.

Remarks. We have allowed the organisation of levels to be very flexible, and thus the effects and levels can be used to complement each other in various different ways. Considering the extreme case of only one level, we return to our original effects proposal from the previous section, in which all the work must be done by the effects. On the other hand, if every class has a level to itself, we essentially impose a total ordering on classes (which may not be possible within our restrictions, for all programs), and no effect annotations are required at all. In practice, we envisage that the levels will be used to separate away previously written library classes from those being currently developed and verified.

6 Extended Technique

So far, static invariants cannot mention the fields of instance objects, and instance invariants cannot mention static fields. It seems reasonable to question whether this is enough. For example, if we wished to write a class MyThread in which each instance object was assigned a unique identifier id, we might like an invariant to express that distinct MyThread objects have different ids[14]. These kinds of invariants involve both static fields and instance fields. It is desirable to extend our technique to handle these more-expressive invariants. We could allow instance invariants to mention static fields (of the same class, and perhaps superclasses) in their invariants. The alternative approach is, instead of enriching instance invariants, to enrich *static* invariants with the ability to quantify over *all* instances of a class. In fact, any instance invariant mentioning static fields can always be expressed as a static invariant by adding a quantified object to replace all the mentions of **this**. However, enriching static invariants in this way can be more general if we allow multiple quantifiers. If we wanted to express the described invariant of MyThread, we could do so by the static invariant forall MyThread o_1, o_2: $o_1 \neq o_2 \Rightarrow o_1$.id$\neq o_2$.id. However, it is not clear how to express this at the level of an instance invariant (without quantifiers).

[14] This is an actual invariant of the Thread class in the Java API.

We choose to add the ability to quantify over fields of instances in static invariants. In static invariants of class c, if o is a quantified object variable, the only fields of o which may be mentioned in the invariants are those declared in class c. This restriction corresponds to the notion of *subclass separation* described for VT (see [10] for details).

Remark. Although it is true that any instance invariant mentioning static fields can be encoded as a static invariant quantifying over instances, this does not quite mean the two are interchangeable with respect to our technique. The reason is that although these invariants express the same properties, because one is an invariant per object, and one is an invariant of the class, they will be expected to hold at different times.

To work out exactly what changes were needed to our technique in order to retain soundness, we were guided by the soundness conditions of [2] (*cf.* Def. 1). Essentially, having made a change to our $\mathbb{D}$ parameter (by changing which invariants can depend on instance fields), the conditions presented there implied the minimal necessary changes to the other parameters of our technique in order to restore soundness. We highlight the differences between the new and the previous technique through the use of italics.

$\mathbb{X}$ invariants of objects (reflexively, transitively) owned by peers, plus all invariants in trees not currently visited on the call stack.
$\mathbb{V}$ invariants of all transitive owners of the current receiver, plus invariants of peers of the current receiver, *and their classes.*
$\mathbb{D}$ Invariants of peers and transitive owners may depend on the field of an object or class. *Additionally, static invariants of the class in which the field is declared.*
$\mathbb{B}$ Before making a method call, *the invariants of the classes of all of the peers of the current receiver must be established.* Furthermore, if the callee is a peer of the current receiver, then the invariants of all peers must be established.
$\mathbb{E}$ the invariants of all visible peers, *and their classes.*
$\mathbb{U}$ A field of an object (or class) may only be assigned to by its owner, or by any of its peers.
$\mathbb{C}$ A call to an instance method is allowed if the callee is a `peer` or `rep` of the current receiver. A call to a static method m on class c is allowed if either the current receiver is c itself, or else c is not on the current call stack.

Soundness. Informally, the soundness of this extended technique follows from the soundness of the previously-presented versions, as follows:

Proposition 2 no longer holds. Namely, because of the extended language of invariants in this new version of our technique, it is possible for many more methods to cause such invariants to break. However, our technique does *not* allow these invariants to remain broken in any more visible states than was previously allowed. Essentially, any invariants which are broken due to the quantification over instances now possible, will always be reestablished at the next visible state (either the end of the method call, or before the next method call; whichever is the sooner). This is reflected in our $\mathbb{B}$ and $\mathbb{E}$ defined above. Therefore, although Proposition 2 does not hold, Theorem 1 can still be proved, essentially because enough extra proof obligations are imposed before a method call takes place.

7 Conclusions, Related Work, and Future Work

We have outlined a verification technique based on VT, catering for static fields, methods, and invariants. In the process, we extended the usual heap topology of ownership types, and tackled potential callbacks through a combination of effects, levels, and the OAM discipline.

Universe types as implemented in JML [5] require static fields to be readonly. JML's static invariants may only refer to static fields, while instance invariants may refer to both static and instance fields [6, Sec. 8.2]. In JML and in our work, both instance and static invariants are supposed to hold in visible states [10]. In JML's universe types, static methods are executed relative to the context of the object who called the static method. This allows one to implement static factory methods, which create new objects in the context of their caller. We can extend our approach to support factory methods by incorporating *ownership transfer* [11], allowing a method to create a new object, but to postpone the decision of assigning it an owner.

In [7], Leino and Müller extend the Boogie methodology [1] to static invariants: static fields may be reps; class invariants may mention static rep fields and also quantify over objects of their class. The callback problem is solved by making explicit the state in which static invariants may be assumed to hold, and by enclosing expressions that potentially break the static invariant of a class in expose blocks. In order to support abstraction in method specifications, a *validity ordering* is used to allow a class to implicitly expect the static invariants of 'smaller' classes. This issue is similar to one of the motivations for introducing our levels. The validity ordering, however, has the side-effect for static initialisation that subclasses be initialised before superclasses.

In Jacobs et al.'s work [4], Spec# annotations are suggested to cater for local reasoning in the presence of multithreading. Again, static fields may be reps, and static invariants may depend on the (transitively) owned objects. Both our system and theirs need to address potential circularities: ours in order to avoid visiting classes in an inconsistent state, and [4] in order to prevent deadlocks. They require a partial ordering of locks, which, in a way, corresponds to our levels. Two locks on the 'same level' are not allowed to be consecutively acquired. In contrast, we permit method calls between classes on the same level, if the effects allow it. Our work may be seen as the visible-states-based counterpart of [4, 7].

We have not discussed static initialisation in this paper. In brief, we expect to be able to incorporate the Java semantics for static initialisation. In terms of our topology, initialisation is best modelled by considering that the tree owned by a class comes into existence at the moment static initialisation of the class begins (and is initially empty, apart from the owning class). Static initialisers may assume all of the invariants of lower levels, and no others (since the restrictions on method calls are not respected by the execution of static initialisers). Exploring these issues in more detail will be the subject of future work. We also plan to complete the formal presentation of our work, and to study class visibility, modularity, readonly fields, pure methods, and factory methods.

Acknowledgements. This paper has been greatly improved by comments and generous feedback from Adrian Francalanza and the (anonymous) FTfJP reviewers. We are also grateful to Nick Cameron, Werner Dietl, and Jayshan Raghunandan for discussions on static methods and verification. This work was funded in part by the IST-2005-015905 MOBIUS project.

References

1. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, LNCS. Springer, 2005.
2. S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, 2008.
3. K. Huizing and R. Kuiper. Verification of object-oriented programs using class invariants. In *FASE*, volume 1783 of *LNCS*, pages 208–221. Springer, 2000.
4. B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs. *Electronic Notes on Theoretical Computer Science special issue on Thread Verification (TV06),*, 174:23–47, 2007.
5. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual—section on Universe annotations, February 2007. `www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_18.html#SEC205`.
6. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Available from `http://www.jmlspecs.org`, May 2008.
7. K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In *Formal Methods*, 2005.
8. Y. Lu, J. Potter, and J. Xue. Object Invariants and Effects. In *ECOOP*, volume 4609 of *LNCS*, pages 202–226. Springer, 2007.
9. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.
10. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.
11. P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 461–478. ACM, 2007.
12. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, 1997.

Checking well-formedness of pure-method specifications

Arsenii Rudich[1], Ádám Darvas[1], and Peter Müller[2]

[1] ETH Zurich, Switzerland, {arsenii.rudich,adam.darvas}@inf.ethz.ch
[2] Microsoft Research, USA, mueller@microsoft.com

Abstract. Contract languages such as JML and Spec# specify invariants and pre- and postconditions using side-effect free expressions of the programming language, in particular, pure methods. For such contracts to be meaningful, they must be well-formed: First, they must respect the partiality of operations, for instance, the preconditions of pure methods used in the contract. Second, they must enable a consistent encoding of pure methods in a program logic, which requires that their specifications are satisfiable and that recursive specifications are well-founded.
This paper presents a technique to check well-formedness of contracts. We give proof obligations that are sufficient to guarantee the existence of a model for the specification of pure methods. We improve over earlier work by providing a systematic solution including a soundness result and by supporting more forms of recursive specifications. Our technique has been implemented in the Spec# programming system.

1 Introduction

Contract languages such as the Java Modeling Language (JML) [21] and Spec# [2] specify invariants and pre- and postconditions using side-effect free expressions of the programming language. While contract languages are natural for programmers, they pose various challenges when contracts are encoded in the logic of a program verifier or theorem prover, especially when contracts use pure (side-effect free) methods [13]. This paper addresses two challenges related to pure-method specifications.

The first challenge is how to ensure that a specification is *well-defined*, that is, that all partial operations are applied within their domain. For instance method calls are well-defined only for non-null receivers and when the precondition of the method is satisfied. This challenge can be solved by encoding partial functions as under-specified total functions [15]. However, it has been argued that such an encoding is counter-intuitive for programmers, is not well-suited for runtime assertion checking, and assigns meaning to bogus contracts instead of having them rejected by a verifier [8]. Another solution is the use of 3-valued logic, such as LPF [3]. However, 3-valued logic is typically not supported by the theorem provers that are used in program verifiers. We present a technique based on 2-valued logic to check whether a specification satisfies all partiality constraints. If the check fails, the specification is rejected.

```
interface Sequence {
  [Ghost] int Length;

  invariant Length >= 0;
  invariant IsEmpty() ==> Length == 0;
  invariant !IsEmpty() ==> Length == Rest().Length + 1;

  [Pure][Measure=Length] int Count(Object c)
     requires !IsEmpty();
     ensures result >= 0;
     ensures result == (GetFirst() == c ? 1 : 0) +
                      (Rest().IsEmpty() ? 0 : Rest().Count(c));
  [Pure] bool IsEmpty();
  [Pure] Object GetFirst()
     requires !IsEmpty();
  [Pure] Sequence Rest()
     requires !IsEmpty();
     ensures result != null;

  // other methods and specifications omitted
}
```

Fig. 1. Specification of interface `Sequence`. We use a notation similar to Spec#, which is an extension of C#. The **`Pure`** attribute marks a method to be side-effect free; pre- and postconditions are attached to methods by **`requires`** and **`ensures`** clauses, respectively. Invariants are specified in **`invariant`** clauses; in postconditions, **`result`** denotes the return value of methods. User-specified recursion measures are given by the **`Measure`** attribute. Fields marked with the **`Ghost`** attribute are specification-only.

The second challenge is how to ensure that a specification is consistent. In order to reason about contracts that contain pure-method calls, pure methods must be encoded in the logic of the program verifier. This is typically done by introducing an uninterpreted function symbol for each pure method m, whose properties are axiomatized based on m's contract and object invariants [10, 13]. A specification is *consistent* if this axiomatization is free from contradictions. Consistency is crucial for soundness. We present a technique to check consistency by showing that the contracts of pure methods are satisfiable and well-founded if they are recursive. If the consistency check fails, the specification is rejected.

An inconsistent specification of a method m is not necessarily detected during the verification of m's implementation [13]: (1) m might be abstract; (2) partial correctness logics allow one to verify m w.r.t. an unsatisfiable specification if m's implementation does not terminate; (3) any implementation could be trivially verified based on inconsistent axioms stemming from inconsistent pure-method specifications; this is especially true for recursion, when the axiom for m is needed to verify its implementation. These reasons justify the need for verifying consistency of specifications independently of implementations.

We illustrate these challenges by the interface `Sequence` in Fig. 1. It contains pure methods to query whether the sequence is empty, and to get the first element and the rest of the sequence. Method `Count` returns the number of occurrences of its parameter in the sequence. The interface contains the specification-only ghost field `Length`, which represents the length of the sequence. The interface is equipped with method specifications and invariants specifying `Length`.

We call a specification *well-formed* if it is well-defined and consistent. The main difficulty in the checking of well-formedness lies in the subtle dependencies between the specification elements. For instance, to be able to show that the expression `Rest().Count(c)` in `Count`'s postcondition is well-defined, the guarding condition `!Rest().IsEmpty()`, the precondition of `Count`, and the contract of `Rest` are needed. These specification elements together allow one to conclude that the receiver is not null and that the preconditions of `Rest` and `Count` are satisfied. That is, we need the specification of (axioms for) some pure methods to prove the well-definedness of other pure methods.

The second challenge is illustrated by the specification of method `Count`. Consistency requires that there actually is a result value for each call to `Count`. This would not be the case, for instance, if the first postcondition required **`result`** to be strictly positive. Since the specification of `Count` is recursive, proving the existence of a result value relies on the specification of `Count`. Using this specification is sound since the recursion in `Count`'s specification is well-founded: the first and third invariant, and the precondition of `Count` guarantee that the sequence is finite, and the guarding condition together with the precondition of `Count` and the third invariant guarantees that we recurse on a shorter sequence. Again, we have a subtle interaction between specifications: proving the consistency of a pure method makes use of the specification of this method as well as invariants and the specification of the methods mentioned in these invariants.

These examples demonstrate that generating the appropriate proof obligations to check well-formedness is challenging. A useful checker must permit dependencies between specification elements, but prevent circular reasoning.

Approach and contributions. We show well-formedness of specifications by posing proof obligations to ensure: (1) that partial operations are applied within their domains, (2) the existence of a possible result value for each pure method, and (3) that recursive specifications are well-founded. In order to deal with dependencies between pure methods, we determine a dependency graph, which we process bottom-up. Thereby, one can use the properties of a method m to prove the proof obligations for the methods using m.

To deal with partiality, we interpret specifications in 3-valued logic. However, we want to support standard theorem provers, which typically use 2-valued logic and total functions [22, 14]. Therefore, we express the proof obligations in 2-valued logic by applying the Δ formula transformer [17] to the specification expressions. We proved the following soundness result: If all proof obligations for the pure methods of a program are proved then there is a partial model for the axiomatization of these pure methods. In other words, we guarantee that the partiality constraints are satisfied and the axiomatization is consistent.

Our approach differs from existing solutions for theorem provers [11, 22], where consistency is typically enforced by restricting specifications to conservative extensions, but no checks are performed for axioms. Since specifications of pure methods are axiomatic, the approach of conservative extensions is not applicable to contract languages. Moreover, theorem provers require the user to resolve dependencies by ordering the elements of a theory appropriately. We determine this order automatically using a dependency graph.

Our approach improves on existing solutions for program verifiers in three ways. First, it supports (mutually) recursive specifications, whereas in previous work recursive specifications are severely restricted [13, 12]. Second, our approach allows us to use the specification of one method to prove well-formedness of another, which is needed in many practical examples. Such dependencies are not discussed in previous work [9, 13] and are not supported by program verifiers that perform consistency checks, such as Spec#. Neglecting dependencies leads to the rejection of well-formed specifications. Third, we prove consistency for the axiomatization of pure methods; such a proof is either missing in earlier work [9, 12] or only presented for a very restricted setting [13].

For simplicity, we consider pure methods to be strongly-pure. That is, pure methods may not modify the heap in any way. An extension to weakly-pure methods [13], which may allocate and initialize objects, is possible.

Outline. Sec. 2 defines well-formedness of pure-method specifications. We present sufficient proof obligations to guarantee the existence of a model in Sec. 3. We discuss how our technique can be applied with automatic theorem provers in Sec. 4. We summarize related work in Sec. 5 and offer conclusions in Sec. 6.

2 Well-formedness

In this section, we define the well-formedness criteria for the specifications of pure methods. Even though some criteria such as partiality also apply to non-pure methods, we focus on pure methods in the following.

Preliminaries. We assume a set **Heap** of heaps with the usual properties. For simplicity, we assume that a program consists of exactly one class; a generalization to several classes and subclassing is possible.

Since there is a one-to-one mapping between pure methods and the corresponding uninterpreted function symbols, we can state the well-formedness criteria directly on the function symbols. In particular, we say "the specification of a function f" to abbreviate "the specification of the pure method encoded by function f". We assume a signature with the function symbols $\mathbf{F} := \{f_1,\ f_2, , \ldots,\ f_n\}$, which correspond to the pure methods of a program. For simplicity we assume pure methods to have exactly one explicit parameter. Thus, all functions in $\mathbf{F}$ are ternary with parameters for the heap (h), receiver object (o), and explicit parameter (p). We assume that all formulas and terms are well-typed.

We define a specification of $\mathbf{F}$ as $\mathbf{Spec} := \langle \mathbf{Pre}, \mathbf{Post}, \mathbf{INV} \rangle$, where:

- **Pre** maps each $f_i \in \mathbf{F}$ to a formula. We denote $\mathbf{Pre}(f_i)$ as $\mathbf{Pre}_{f_i}$. Due to the syntactic structure of preconditions, the only free variables in $\mathbf{Pre}_{f_i}$ are h, o, and p.
- **Post** maps each $f_i \in \mathbf{F}$ to a formula. We denote $\mathbf{Post}(f_i)$ as $\mathbf{Post}_{f_i}$. Due to the syntactic structure of postconditions, the only free variables in $\mathbf{Post}_{f_i}$ are h, o, p, and the result variable res. Since we assume pure methods to be strongly-pure, one heap variable is enough to capture the heap before and after the method execution.
- **INV** is a set of formulas $\{\mathbf{Inv}_1, \mathbf{Inv}_2, \ldots, \mathbf{Inv}_m\}$. Due to the syntactic structure of invariants, the only free variables in $\mathbf{Inv}_i \in \mathbf{INV}$ are the heap h and the object o to which the invariant is applied.
  We use $\mathbf{SysInv} := \forall\, o \in h. \wedge_{i=1}^{m} \mathbf{Inv}_i$ to denote the conjunction of all invariants for all allocated objects, where $o \in h$ expresses that a reference o refers to an allocated object in heap h. Note that **SysInv** is an open formula with free variable h.

Structures and interpretations. To define the interpretation of specifications, we use a structure $\mathbf{M} := \langle \mathbf{Heap}, \mathbf{R}, \mathbf{I} \rangle$, where $\mathbf{R}$ is the set of references and $\mathbf{I}$ is an interpretation function for the specification of a function $f \in \mathbf{F}$: $\mathbf{I}(f) : \mathbf{Heap} \times \mathbf{R} \times \mathbf{R} \to \mathbf{R}$. This structure can be trivially extended to other sorts like integer or boolean.

For a formula φ, we define the interpretation in total structures $[\varphi]^2_{\mathbf{M}} e$ in the standard way. Here, e is a *variable assignment* that maps the free variables of φ to values. For the interpretation in partial structures $[\varphi]^3_{\mathbf{M}} e$, we follow Berezin et al. [5]: intuitively, the interpretation of a function is defined if and only if the interpretations of all parameters are defined and the vector of parameters belongs to the function domain. The interpretation of logical operators and quantifiers is defined according to Kleene logic [20].

A total interpretation maps a formula to a value in $\mathbf{Bool}_2 := \{\mathbf{T}, \mathbf{F}\}$, while a partial interpretation maps a formula to a value in $\mathbf{Bool}_3 := \{\mathbf{T}, \mathbf{F}, \perp\}$. A partial structure $\mathbf{M}$ can be extended to a total structure $\hat{\mathbf{M}}$ by defining values of functions outside of their domains by arbitrary values. To check whether or not a value in $\mathbf{Bool}_3$ is $\perp$ we use the following function:

$$\mathbf{wd} : \mathbf{Bool}_3 \to \mathbf{Bool}_2$$

$$\mathbf{wd}(x) := \begin{cases} \mathbf{T}\ , & \text{if } x \in \{\mathbf{T}, \mathbf{F}\} \\ \mathbf{F}\ , & \text{if } x = \perp \end{cases}$$

Well-formedness criteria. A specification **Spec** is well-formed (denoted by $\models \mathbf{Spec}$) if there exists a partial model $\mathbf{M}$ for the specification. A structure $\mathbf{M}$ is a *partial model* for specification **Spec**, denoted by $\mathbf{M} \models \mathbf{Spec}$, if it satisfies the following four criteria:

1. Invariants are never interpreted as $\bot$, that is, for each $\mathbf{heap} \in \mathbf{Heap}$:
   $\mathbf{wd}([\mathbf{SysInv}]^3_M e)$ holds
   where $e := [h \to \mathbf{heap}]$.
2. Preconditions are never interpreted as $\bot$ in heaps that satisfy the invariants of all allocated objects, that is, for each $f \in \mathbf{F}$, $\mathbf{heap} \in \mathbf{Heap}$, $\mathbf{this} \in \mathbf{heap}$, and $\mathbf{par} \in \mathbf{heap}$:
   if $[\mathbf{SysInv}]^3_M e$ holds, then $\mathbf{wd}([\mathbf{Pre}_f]^3_{\mathbf{M}} e)$ holds
   where $e := [h \to \mathbf{heap}, o \to \mathbf{this}, p \to \mathbf{par}]$.
3. The values of the parameters belong to the domain of the interpretation of function symbols, provided that the heap satisfies the invariants and the precondition holds. That is, for each $f \in \mathbf{F}$, $\mathbf{heap} \in \mathbf{Heap}$, $\mathbf{this} \in \mathbf{heap}$, and $\mathbf{par} \in \mathbf{heap}$:
   if $[\mathbf{SysInv}]^3_M e$ and $[\mathbf{Pre}_f]^3_{\mathbf{M}} e$ hold,
   then $\langle \mathbf{heap}, \mathbf{this}, \mathbf{par} \rangle \in \mathbf{dom}(\mathbf{I}(f))$ holds
   where $e := [h \to \mathbf{heap}, o \to \mathbf{this}, p \to \mathbf{par}]$.
4. Postconditions are never interpreted as $\bot$ for any result, and the interpretation of function f as result value satisfies the postcondition, provided that the heap satisfies the invariants and the precondition holds. That is, for each $f \in \mathbf{F}$, $\mathbf{heap} \in \mathbf{Heap}$, $\mathbf{this} \in \mathbf{heap}$, and $\mathbf{par} \in \mathbf{heap}$:
   if $[\mathbf{SysInv}]^3_M e$ and $[\mathbf{Pre}_f]^3_{\mathbf{M}} e$ hold,
   then for each $\mathbf{result} \in \mathbf{heap}$ $\mathbf{wd}([\mathbf{Post}_f]^3_{\mathbf{M}} e')$ holds,
   and $[\mathbf{Post}_f]^3_{\mathbf{M}} e$ holds
   where $e := [h \to \mathbf{heap}, o \to \mathbf{this}, p \to \mathbf{par}, res \to \mathbf{I}(f)(\mathbf{heap}, \mathbf{this}, \mathbf{par})]$,
   $e' := [h \to \mathbf{heap}, o \to \mathbf{this}, p \to \mathbf{par}, res \to \mathbf{result}]$.

Axiomatization. As motivated in Sec. 1, a verification system needs to extract axioms from the specifications of pure methods. We denote the axiom for function symbol f as $\mathbf{Ax}_f$ and the axioms for all functions as $\mathbf{Ax}_{\mathbf{Spec}}$. Formally:

$$\mathbf{Ax}_f := \forall\, h, o \in h, p \in h.\ \mathbf{SysInv} \wedge \mathbf{Pre}_f \Rightarrow \mathbf{Post}_f[f(h,o,p)/res]$$

$$\mathbf{Ax}_{\mathbf{Spec}} := \bigwedge_{f \in \mathbf{F}} \mathbf{Ax}_f$$

From well-formedness criterion 4 and $\mathbf{Ax}_f$, we can conclude that if a structure $\mathbf{M}$ is a partial model for specification $\mathbf{Spec}$ then it is a model for $\mathbf{Ax}_{\mathbf{Spec}}$:

$$\text{if} \quad \mathbf{M} \models \mathbf{Spec} \quad \text{then} \quad \mathbf{M} \models \mathbf{Ax}_{\mathbf{Spec}}$$

Consequently, if specification $\mathbf{Spec}$ is well-formed then the axioms are consistent:

$$\text{if} \quad \models \mathbf{Spec} \quad \text{then} \quad \models \mathbf{Ax}_{\mathbf{Spec}}$$

Important to note is that this property does not hold in the other direction, that is, if $\models \mathbf{Ax}_{\mathbf{Spec}}$ then $\models \mathbf{Spec}$ is not necessarily true. For example, consider a method with precondition `1/0 == 1/0` and postcondition `true`. In 2-valued logic, the axiom is trivially consistent, but the specification is not well-formed (criterion 2). This demonstrates that our well-formedness criteria require more than just consistency, namely also satisfaction of partiality constraints.

3 Checking well-formedness

In this section, we present sufficient proof obligations that ensure that a specification is well-formed, that is, the existence of a model.

3.1 Partiality

We want our technique to work with first-order logic theorem provers, which are often used in program verifiers. These provers check that a formula holds for all total models. However, we need to check properties of partial models. Therefore, we apply a technique that reduces the 3-valued domain to a 2-valued domain by ensuring that $\perp$ is never encountered. This is a standard technique applied in different tools, for instance, in B [4], CVC Lite [5], and ESC/Java2 [9].

The main idea is to use the formula transformer $\varDelta$ [17, 4], which takes a (possibly open) formula φ and *domain restriction* δ, and produces a new formula φ'. The interpretation of φ' in 2-valued logic is true if and only if the interpretation of φ in 3-valued logic is different from $\perp$. The domain restriction δ is a mapping from a set of function symbols $\mathbf{F}_\delta$ to formulas. δ characterizes the domains of the function symbols of $\mathbf{F}_\delta$. For instance for the division operator, the domain restriction δ requires the divisor to be non-zero. Thus, $\varDelta(a/b > 0, \delta) \equiv b \neq 0$.

For lack of space, we do not give the details of the $\varDelta$ operator and refer the reader to [4]. The most important property for our purpose is the following [5]:

$$\mathbf{M} \models \delta \;\Rightarrow\; (\, [\varDelta(\varphi, \delta)]^2_{\hat{M}} e = \mathbf{wd}([\varphi]^3_M e)\,) \qquad (1)$$

which captures the intuition of $\varDelta$ described above. $\varDelta$ is a syntactical characterization of the semantical operation $\mathbf{wd}$. Thus, using $\varDelta$, we can check in 2-valued logic the partiality properties we are interested in.

Property (1) interprets formulas w.r.t. a structure $\mathbf{M}$. This structure with function symbols $\mathbf{F}_\delta$ has to be a model for δ (denoted by $\mathbf{M} \models \delta$), that is:

- The domain formulas are defined, that is, for each $f \in \mathbf{F}_\delta$

  $\mathbf{wd}([\delta(f)]^3_M e)$ holds for all e.
- δ characterizes the domains of function interpretations for $\mathbf{M}$, that is, for each $f \in \mathbf{F}_\delta$ and $\mathbf{val}_1, \ldots, \mathbf{val}_k \in \mathbf{R}$:

  $[\delta(f)]^3_{\mathbf{M}} e$ holds if and only if $\langle \mathbf{val}_1, \ldots, \mathbf{val}_k \rangle \in \mathbf{dom}(\mathbf{I}(f))$

  where $e := [v_1 \to \mathbf{val}_1, \ldots, v_k \to \mathbf{val}_k]$ and $\{v_1, \ldots, v_k\}$ are the parameter names of f. (Since methods have only one explicit parameter, $k = 3$.)

3.2 Incremental construction of model

In general, showing the existence of a model requires one to prove the existence of all its functions. To be able to work with first-order logic theorem provers, we approximate this second-order property in first-order logic. We generate proof obligations whose validity in 2-valued first-order logic guarantees the existence of a model. However, if we fail to prove them then we do not know whether a model exists or not. That is, the procedure is sound but not complete. However, it works for the practical examples we have considered so far.

The basic idea of our procedure is to construct a model incrementally. We build a dependency graph whose nodes are function symbols and invariants. There is an edge from node a to node b if the specification of function a or the invariant a applies function b. The dependency graph of interface `Sequence` is presented in Fig. 2.

The dependency graph may be cyclic. However, we disallow cycles that are introduced by preconditions. In other words, a precondition must not be recursive in order to avoid fix-point computation to define the domain of the function. This is not a limitation for practical examples.

We construct the model by traversing the dependency graph bottom-up. We start with the empty specification $\mathbf{Spec}_0 := \langle \emptyset, \emptyset, \emptyset \rangle$, for which we trivially have a model $\mathbf{M}_0$. In each step j, we select a set of nodes $G_j := \{g_1, g_2, \dots, g_k\}$ such that if there is an edge from g_i to a node n then either n has already been visited in some previous step (i.e., $n \in G_1 \cup ... \cup G_{j-1}$) or $n \in G_j$. Moreover, we choose G_j such that it has one of the following forms:

1. G_j contains exactly one invariant $\mathbf{Inv}_l \in \mathbf{INV}$.
2. G_j contains exactly one function symbol $f_l \in \mathbf{F}$ and the specification of f_l is not recursive.
3. G_j is a set of function symbols, and the nodes in G_j form a cycle in the dependency graph, that is, they are specified recursively in terms of each other. G_j may contain only one node in case of direct recursion.

We call the pre- and postconditions and the invariants of G_j the *current specification fragment*, s_j. We extend $\mathbf{Spec}_{j-1}$ with s_j resulting in $\mathbf{Spec}_j$. We impose proof obligations on s_j that guarantee that the model $\mathbf{M}_{j-1}$ for $\mathbf{Spec}_{j-1}$ can be extended to a model $\mathbf{M}_j$ for $\mathbf{Spec}_j$. Since this construction is inductive, we may assume that all specification fragments processed up to step $j-1$ are well-formed.

It is easy to see that an order in which one can traverse the dependency graph always exists. However, the chosen order may influence the success of the model construction. Essentially one should choose an invariant node whenever possible because the invariant provides information that might be useful for later steps.

3.3 Proof obligations

We now present the proof obligations for the three different kinds of current specification fragments s_j. We refer to the elements of $\mathbf{Spec}_j$ as $\mathbf{Pre}_j$, $\mathbf{Post}_j$, and $\mathbf{INV}_j$. To make the formulas more readable we use the following notations:

- $\mathbf{SysInv}_j := \forall\ o \in h.\ \bigwedge_{Inv \in \mathbf{INV}_j} Inv$. $\mathbf{SysInv}_j$ is the conjunction of invariants processed up to step j. After the last step z of the construction of the model, we have $\mathbf{SysInv}_z = \mathbf{SysInv}$.
- F_j denotes the set of function symbols whose pre- and postconditions have been processed up to step j: $F_j := \mathbf{dom}(\mathbf{Pre}_j)$, and thus $F_j = \mathbf{dom}(\mathbf{Post}_j)$.

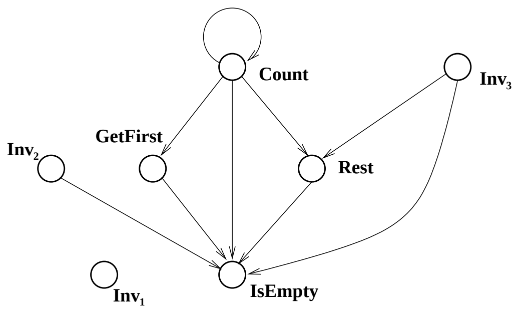


Fig. 2. Dependency graph for interface `Sequence`.

– We denote the axioms for $\mathbf{Spec}_j$ as follows:

$$\mathbf{Ax}^j_f := \forall\ h, o \in h, p \in h.\ \mathbf{SysInv}_j \wedge \mathbf{Pre}_f \Rightarrow \mathbf{Post}_f[f(h,o,p)/res]$$

$$\mathbf{Ax}_{\mathbf{Spec}_j} := \bigwedge_{f \in F_j} \mathbf{Ax}^j_f$$

$\mathbf{Ax}^j_f$ is the definition of the axiom for a function f according to specification $\mathbf{Spec}_j$. Note that the axiom $\mathbf{Ax}^j_f$ may be different for different j since $\mathbf{SysInv}_j$ gets gradually strengthened during the construction of the model. Therefore, the axiom $\mathbf{Ax}^j_f$ becomes gradually weaker. This is an important observation for the soundness of our approach. After the last step z of the construction of the model, we have $\mathbf{Ax}^z_f = \mathbf{Ax}_f$ and $\mathbf{Ax}_{\mathbf{Spec}_z} = \mathbf{Ax}_{\mathbf{Spec}}$.

The following proof obligations are posed on the three different types of specification fragments in step j.

Invariant $\mathbf{Inv}_l$. The invariant $\mathbf{Inv}_l$ must be well-defined for each object, provided the invariants $\mathbf{SysInv}_{j-1}$ hold.

$$\mathbf{Ax}_{\mathbf{Spec}_{j-1}} \Rightarrow \forall\ h.\ (\mathbf{SysInv}_{j-1} \Rightarrow \Delta(\forall\ o \in h.\ \mathbf{Inv}_l, \mathbf{Pre}_{j-1})) \tag{2}$$

Note that we use preconditions $\mathbf{Pre}_{j-1}$ as domain restriction. Although invariants additionally restrict the domain of functions, these restrictions are never violated due to the assumption that $\mathbf{SysInv}_{j-1}$ holds.

Example. We instantiate the proof obligation for a specification fragment from Fig. 1. The corresponding dependency graph is presented in Fig. 2. The traversal of the dependency graph first visits the first invariant since it has no dependencies. The well-definedness of the invariant is trivial. Next, the traversal takes method `IsEmpty`, which is also processed trivially since the method has no specifications. As third node, the second invariant is picked. For this specification fragment, the following proof obligation is generated.

$$\begin{aligned}&\forall\ h.\ ((\forall\ o \in h.\ h[o, \mathit{Length}] \geq 0) \Rightarrow \\ &\quad \Delta(\forall\ o \in h.\ \mathit{IsEmpty}(h,o) \Rightarrow h[o, \mathit{Length}] = 0, \{\langle \mathit{IsEmpty}, \mathit{true}\rangle\}))\end{aligned}$$

where $h[o, f]$ denotes field access with receiver object o and field f in heap h. Note that $\mathbf{Ax}_{\mathbf{Spec}_2}$ has been omitted since it is equivalent to true. After the application of the Δ operator, the proof obligation requires one to prove that (1) o is non-null since it is the receiver of a method call and a field access, and that (2) the domain restriction of *IsEmpty* is not violated. The first property holds since $o \in h$, the second since the domain restriction of *IsEmpty* is true. $\square$

Pre- and postcondition of a single function f_l. This case requires two proof obligations for the non-recursive pre- and postcondition of f_l, respectively. The first proof obligation checks that the precondition of f_l is defined for all receiver objects and parameters in all heaps in which the invariants hold.

$$\mathbf{Ax}_{\mathbf{Spec}_{j-1}} \Rightarrow \forall\ h, o \in h, p \in h.\ (\mathbf{SysInv}_{j-1} \Rightarrow \Delta(\mathbf{Pre}_{f_l}, \mathbf{Pre}_{j-1})) \tag{3}$$

Example. Assume method `Rest` is selected as fourth specification fragment. The corresponding proof obligation is the following.

$$\begin{array}{l} \forall\ h, o \in h. \\ \quad (\,(\forall\ o \in h.\ h[o, \mathit{Length}] \geq 0\ \wedge\ (\mathit{IsEmpty}(h, o) \Rightarrow h[o, \mathit{Length}] = 0)) \Rightarrow \\ \quad\quad \Delta(\neg \mathit{IsEmpty}(h, o), \{\langle \mathit{IsEmpty}, \mathit{true} \rangle\})\,) \end{array}$$

Again, $\mathbf{Ax}_{\mathbf{Spec}_3}$ has been omitted since it is equivalent to true. After the application of the Δ operator, the same properties need to be proven as above: o is non-null and the domain restriction of *IsEmpty* is not violated. $\square$

The second proof obligation checks that the postcondition of f_l is never interpreted as $\perp$ for any result, and that there exists a value which satisfies the postcondition for all receiver objects and parameters that satisfy the precondition in all heaps in which the invariants hold.

$$\begin{array}{l} \mathbf{Ax}_{\mathbf{Spec}_{j-1}} \Rightarrow \forall\ h, o \in h, p \in h.\ (\,\mathbf{SysInv}_{j-1}\ \wedge\ \mathbf{Pre}_{f_l}\ \Rightarrow \\ \quad\quad (\forall\ \mathit{res}.\ \Delta(\mathbf{Post}_{f_l}, \mathbf{Pre}_{j-1})) \wedge (\exists\ \mathit{res}.\ \mathbf{Post}_{f_l})\,) \end{array} \tag{4}$$

Example. The proof obligation for the postcondition of method `Rest` is:

$$\begin{array}{l} \forall\ h, o \in h. \\ \quad (\,(\forall\ o \in h.\ h[o, \mathit{Length}] \geq 0\ \wedge\ (\mathit{IsEmpty}(h, o) \Rightarrow h[o, \mathit{Length}] = 0)) \wedge \\ \quad\quad \neg \mathit{IsEmpty}(h, o) \\ \quad\quad\quad \Rightarrow \\ \quad\quad (\forall\ \mathit{res}.\ \Delta(\mathit{res} \neq \mathit{null}, \{\langle \mathit{IsEmpty}, \mathit{true} \rangle\}))\ \wedge\ (\exists\ \mathit{res}.\ \mathit{res} \neq \mathit{null})\,) \end{array}$$

As before, $\mathbf{Ax}_{\mathbf{Spec}_3}$ is equivalent to true. The first conjunct is proved trivially since formula $\mathit{res} \neq \mathit{null}$ does not contain any partial operation. To satisfy the second conjunct, we instantiate res with o. $\square$

Pre- and postconditions of a set of recursively-specified functions. This case handles both direct and mutual recursion. That is, we have a set of functions $G_j := \{g_1, g_2, \ldots, g_k\}$ with $k \geq 1$. We assume that for each function g_i in G_j the programmer provides a measure function $\| \cdot \|_{g_i} : \mathbf{Heap} \times \mathbf{R} \times \mathbf{R} \rightarrow \mathbb{N}$ using the `Measure` attribute. We assume that there is no recursion via measure functions, that is, the definition of measure function $\| \cdot \|_{g_i}$ may only contain function symbols from $G_1 \cup \ldots \cup G_{j-1}$, but not from G_j.

Since preconditions must not be recursively specified (see Sec. 3.2), the proof obligation for the precondition of each g_i is identical to proof obligation (3) for the non-recursive case.

In order to prove well-formedness of postconditions, we first need to show that user-specified measures are well-defined and non-negative. For a function g_i with measure attribute `Measure=`μ_{g_i}, we introduce a new pure method M_{g_i} with precondition $\mathbf{Pre}_{g_i}$ and postcondition $\mu_{g_i} \geq 0$. The dependency graph is extended with a node for M_{g_i} and an edge from g_i to M_{g_i}. Node M_{g_i} is processed like any other node. This allows measures to rely on invariants and to contain calls to pure methods.

Proof obligation (5) below for postconditions is similar to proof obligation (4), but differs in two ways: First, we have to prove that the recursive specification is well-founded. Since we have already shown that our measure functions yield non-negative numbers, it suffices to show that the measure decreases for each recursive application. We achieve this by using a domain restriction that additionally requires the measure for recursive applications to be lower than the measure *ind* of the function being specified. If the measure *ind* is 0, the domain restriction becomes false, which prevents further recursion. Note that the occurrence of *ind* seems to violate the condition that domain restrictions do not contain free variables other than the parameters of the function whose domain they characterize. However, since *ind* is universally quantified, we may consider *ind* to be a constant for each particular application of the domain restriction. (One could think of the universal quantification as an unbounded conjunction, where *ind* is a constant in each of the conjuncts.)

Second, for the proof of well-formedness of the specification of a function g_i, we may assume the properties of the functions recursively applied in this specification. This is an induction scheme over the measure *ind*, which is expressed by the assumption in lines 4 and 5 of the following proof obligation, which must be shown for each method g_i.

$$\begin{array}{l}
\mathbf{Ax}_{\mathbf{Spec}_{j-1}} \Rightarrow \\
\quad \forall\ ind \in \mathbb{N}, h, o \in h, p \in h. \\
\qquad (\ \mathbf{SysInv}_{j-1}\ \wedge\ \mathbf{Pre}_{g_i}\ \wedge\ \|\langle h, o, p \rangle\|_{g_i} = ind\ \wedge \\
\qquad\quad (\ \bigwedge_{l=1}^{k} \forall\ o' \in h, p' \in h.\ \mathbf{Pre}_{g_l}[o'/o, p'/p]\ \wedge\ \|\langle h, o', p' \rangle\|_{g_l} < ind \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{Post}_{g_l}[o'/o, p'/p, g_l(h, o', p')/res]\ ) \\
\qquad\qquad \Rightarrow \\
\qquad\quad (\forall\ res.\ \Delta(\mathbf{Post}_{g_i}, \mathbf{Pre}_{j-1} \cup \{\langle g_l, \mathbf{Pre}_{g_l} \wedge \|\langle h, o, p \rangle\|_{g_l} < ind \rangle \mid l \in 1..k\})) \wedge \\
\qquad\quad (\exists\ res.\ \mathbf{Post}_{g_i})\ )
\end{array} \qquad (5)$$

Example. Since the size of proof obligation (5) for the postcondition of method `Count` (the only recursive specification in our example) is rather large, we use a considerably smaller example here, namely the factorial function with the following specification.

```
[Pure][Measure=p] int Fact(int p)
  requires p >= 0;
  ensures p == 0 ==> result == 1;
  ensures p > 0 ==> result == Fact(p-1)*p;
```

To simplify the example, we omit the variables for heap h and receiver object o.

First, we need to prove that measure p is well-defined and non-negative. This is trivially proven since the measure does not contain partial operators and the precondition of `Fact` guarantees that p is non-negative.

Next, we need to show proof obligation (5). For brevity, we only show it for the second postcondition, which is the interesting case containing recursion:

$$
\begin{array}{l}
\forall\ ind \in \mathbb{N}, p. \\
\quad (\, p \geq 0 \ \wedge\ p = ind \wedge \\
\quad\ (\forall\ p'.\ p' \geq 0 \wedge p' < ind\ \Rightarrow \\
\qquad\quad (p' = 0 \Rightarrow Fact(p') = 1\,)\ \wedge\ (p' > 0 \Rightarrow Fact(p') = Fact(p' - 1) * p')\,) \\
\qquad \Rightarrow \\
\quad\ (\forall\ res.\ \Delta(\, p > 0 \Rightarrow res = Fact(p-1) * p, \{\langle Fact,\ p \geq 0 \wedge p < ind\rangle\ \})) \wedge \\
\quad\ (\exists\ res.\ p > 0 \Rightarrow res = Fact(p-1) * p)\,)
\end{array}
$$

We need to show that the two quantified conjuncts on the right-hand side of the implication hold. Proving that the existential holds is straightforward due to the equality. The other conjunct is more interesting. The only partial operator is $Fact$ and after applying the Δ operator the sub-formula simplifies to:

$$\forall\ res.\ p > 0 \Rightarrow p - 1 \geq 0\ \wedge\ p - 1 < ind$$

The first conjunct is provable from $p > 0$ and the second from $p = ind$ in the premise of the proof obligation. □

Soundness. The above proof obligations are sufficient to show that a specification is well-formed:

> **Theorem.** If a specification **Spec** does not contain recursive preconditions and all of the above proof obligations for **Spec** hold then **Spec** is well-formed, that is, $\models$ **Spec** holds.

The proof of this theorem runs by induction on the order of specification fragments given by the dependency graph. For each recursive specification fragment, the proof uses a nested induction on the recursion depth *ind*. Due to lack of space, we refer to [23] for a detailed proof sketch.

Modularity. In general, adding new classes to a program does not invalidate the proofs for the well-formedness criteria of existing methods and invariants. This is because we assume behavioral subtyping, which ensures that the axiom for an overriding method is weaker than the axiom for the overridden method. Although new classes can introduce cycles in the dependency graph that involve existing methods, proofs remain valid since we introduce new function symbols for overriding methods, which thus do not interfere with existing proofs.

The invariants of additional classes strengthen **SysInv**, which appears as part of the premises of proof obligations; thus, they weaken the proof obligations.

4 Application with automatic theorem provers

The proof obligations presented in the previous section are sufficient to show the well-formedness of a specification. However, they are not well-suited for automatic theorem provers such as Simplify [14] or Z3 for two reasons. First, the proof obligations to ensure consistency for postconditions (proof obligations (4) and (5)) contain existential quantifiers, for which automatic theorem provers often do not find suitable instantiations. Second, the proof obligation for the well-foundedness of recursive specifications (proof obligation (5)) is in general proved by induction on *ind*, but induction is not supported well by automatic theorem provers. In this section, we discuss these issues.

Consistency. Spec# uses four approaches to find witnesses for the satisfiability of a specification, that is, instantiations for the existential quantifiers[1]. First, if a postcondition has the form **`result`** `R E`, where `R` is a reflexive operator and `E` is an expression that does not contain **`result`** and recursive calls, then there always exists a possible result value, namely, the value of `E` [12]. Thus, this part of the proof obligations can be dropped. Second, if a pure method has a body of the form **`return`** `E`, where `E` does not contain a recursive call, then expression `E` is a likely candidate for a witness. It suffices to use a simplified proof obligation to show that this candidate actually is a witness. Third, for many postconditions, good candidates for witnesses can be inferred by simple heuristics. For instance, for a postcondition **`result`** `> E`, one might try `E + 1`. Finally, if the former approaches do not work, Spec# allows programmers to specify witnesses for model fields explicitly. One could use the same approach for pure methods.

Well-foundedness. Proof obligation (5) in general requires induction. For instance, if function $f(n)$ has a postcondition $(n = 0 \Rightarrow res = 1) \wedge (n > 0 \Rightarrow res = 1/f(n-1))$, one needs to apply induction to prove that f never returns zero. However, induction is needed only if the function is specified recursively *and* the recursive call occurs as an argument to a partial function, as in this example. In our experience, this is not the case for most specifications. For instance, proving proof obligation (5) for the factorial function does not require induction, as we have shown in Sec. 3.3. Therefore, this proof obligation is not a major limitation in practice.

[1] Most of these approaches were proposed and implemented by Rustan Leino and Ronald Middelkoop.

5 Related work

We sketch what three important groups of formal systems do in the areas of consistency and well-definedness checking.

Theorem provers. Isabelle [22] is an interactive LCF-style theorem proving framework based on a small logical core. Everything on top of the core is supposed to be defined by conservative extensions, which ensures the consistency of the specification. The use of axioms is possible but discouraged since inconsistency may be introduced. Recursion (both direct and mutual) is supported and the well-foundedness of the recursion has to be proven. Isabelle handles partiality by under-specification [15] and requires no well-definedness checks.

PVS [11] is similar to Isabelle with respect to consistency guarantees. The main difference is in the modeling of partial functions. Although PVS also considers functions to be total, predicate subtyping is used to restrict the domain of functions. This makes the type system undecidable leading to Type Correctness Conditions to be proven [24].

Formal software development systems. Z is a formal specification language for computing systems [25]. The work closest to ours is the approach of Hall et al., which shows how a model conjecture can be derived from a Z specification [16]. Partiality is handled by under-specification [26].

The B method [1] is similar to Z but is more focused on the notion of refinement. Satisfiability of the specification has to be proven in each refinement step. B allows users to add axioms whose consistency is not checked. Thus, they may introduce unsoundness. B allows functions to be partial and requires specifications to be well-defined by using the Δ formula transformer [4].

VDM [18] also checks satisfiability of specifications and allows the use of (possibly inconsistent) axioms. VDM uses LPF [3], a 3-valued logic. In contrast to our approach, well-definedness is not proven before the actual proof process, but is proven together with the validity of verification conditions.

Program verifiers. ESC/Java2 [19] is an automatic extended static checker for Java programs annotated with JML specifications. The tool axiomatizes specifications of pure methods [10]. Consistency of the axiom system is not ensured, which can lead to unsoundness. Recently, well-definedness checks have been added by Chalin [9] but it is not clear how dependencies among specification elements are handled, and no soundness proof is provided.

Jack [7] is a program verifier for JML annotated Java programs. The backend prover of the tool is Coq [6]. The tool axiomatizes pre- and postconditions of pure methods separately. This separation ensures that axioms are only instantiated when a pure-method call occurs in a given verification condition—as opposed to be available to the theorem prover at any time. However, since Jack does not check consistency, unsoundness can still occur by the use of axioms. Jack does not support mutual recursion and does not check well-definedness.

The Spec# program verifier ensures consistency of axioms over pure methods by the approaches described in Sec. 4 and by allowing programmers to declare a static call-order on pure methods. Only a simple form of recursive specifications is supported where the measure is based on the ownership relation. The well-foundedness of this relation can be checked by the compiler without proof obligations [12]. Spec# does not fully check well-definedness of specifications.

Our technique improves on our own earlier work [13] by allowing pure-method calls in invariants, ensuring well-formedness of specifications, supporting mutual recursion, taking dependencies into account, and by precisely defining what the proposed proof obligations guarantee. On the other hand, [13] handles weak-purity which we omitted in this paper for simplicity. However, our work could be extended following the technique described in [13].

6 Conclusion

Well-formedness of specifications is important to meet programmer expectations, to reconcile static and runtime assertion checking, and to ensure soundness of static verification. We presented a new technique to check the well-formedness of specifications. We showed how to incrementally construct a model for the specification, which guarantees that the partiality constraints of operations are respected and that the axiomatization of pure methods is consistent. Our technique can be applied in any verification system, regardless of its contract language, logic, or backend theorem prover. As a proof of concept, we implemented our technique in the Spec# verification system.

As future work, we plan to develop adapted proof obligations that require induction in fewer cases. We expect that this can be done by generating specific proof obligations for each given recursive call, which encode the inductive argument. We also plan to investigate how to conveniently specify measures for methods that traverse object structures.

Acknowledgments. We are grateful to Julien Charles, Farhad Mehta, and Burkhart Wolff for helpful discussions on related work. Thanks also to the anonymous reviewers for their insightful comments. Geraldine von Roten implemented the presented technique in the Spec# system.

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

1. J. R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.

3. H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.
4. P. Behm, L. Burdy, and J.-M. Meynadier. Well Defined B. In *International B Conference*, pages 29–45. Springer-Verlag, 1998.
5. S. Berezin, C. Barrett, I. Shikanian, M. Chechik, A. Gurfinkel, and D. L. Dill. A practical approach to partial functions in CVC Lite. In *PDPAR*, 2004.
6. Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer-Verlag, 2004.
7. L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In *FME*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, 2003.
8. P. Chalin. Are the logical foundations of verifying compiler prototypes matching user expectations? *Formal Aspects of Computing*, 19(2):139–158, 2007.
9. P. Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *ICSE*, pages 23–33. IEEE Computer Society, 2007.
10. D. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, October 2005.
11. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. *A Tutorial Introduction to PVS*, April 1995.
12. Á. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, volume 4422 of *LNCS*, pages 336–351. Springer-Verlag, 2007.
13. Á. Darvas and P. Müller. Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology (JOT)*, 5(5):59–85, June 2006.
14. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.
15. D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*, pages 366–373. Springer-Verlag, 1995.
16. J. G. Hall, J. A. McDermid, and I. Toyn. Model conjectures for Z specifications. In *7th International Conference on Putting into Practice Methods and Tools for Information System Design*, pages 41–51, 1995.
17. A. Hoogewijs. On a formalization of the non-definedness notion. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 25:213–217, 1979.
18. C. B. Jones. *Systematic software development using VDM.* Prentice Hall, 1986.
19. J. R. Kiniry and D. R. Cok. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS*, volume 3362 of *LNCS*, pages 108–128. Springer-Verlag, 2005.
20. S. C. Kleene. *Introduction to Metamathematics.* Van Nostrand, 1952.
21. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
22. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS.* Springer-Verlag, 2002.
23. A. Rudich, Á. Darvas, and P. Müller. Checking well-formedness of pure-method specifications (Full Paper). Technical Report 588, ETH Zurich, 2008.
24. J. Rushby, S. Owre, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.
25. J. M. Spivey. *Understanding Z: a specification language and its formal semantics.* Cambridge University Press, 1988.
26. S. H. Valentine. Inconsistency and Undefinedness in Z - A Practical Guide. In *International Conference of Z Users*, pages 233–249. Springer-Verlag, 1998.

Efficient Well-Definedness Checking

Ádám Darvas, Farhad Mehta, and Arsenii Rudich

ETH Zurich, Switzerland,
{adam.darvas,farhad.mehta,arsenii.rudich}@inf.ethz.ch

Abstract. Formal specifications often contain partial functions that may lead to ill-defined terms. A common technique to eliminate ill-defined terms is to require well-definedness conditions to be proven. The main advantage of this technique is that it allows us to reason in a two-valued logic even if the underlying specification language has a three-valued semantics. Current approaches generate well-definedness conditions that grow exponentially with respect to the input formula. As a result, many tools prove shorter, but stronger approximations of these well-definedness conditions instead.
We present a procedure which generates well-definedness conditions that grow linearly with respect to the input formula. The procedure has been implemented in the Spec# verification tool. We also present empirical results that demonstrate the improvements made.

1 Introduction

Formal specifications often allow terms to contain applications of partial functions, such as division $x \,/\, y$ or factorial $\mathit{fact}(z)$. However, it is not clear what value $x \,/\, y$ yields if y is 0, or what value $\mathit{fact}(z)$ yields if z is negative. Specification languages need to handle *ill-defined terms*, that is, either have to define the semantics of partial-function applications whose arguments fall outside their domains or have to eliminate such applications.

One of the standard approaches to handle ill-defined terms is to define a three-valued semantics [22] by considering ill-defined terms to have a special value, *undefined*, denoted by $\perp$. That is, both $x \,/\, 0$ and $\mathit{fact}(-5)$ are considered to evaluate to $\perp$. In order to reason about specifications with a three-valued semantics, undefinedness is lifted to formulas by extending their denoted truth values to $\{\mathbf{true}, \mathbf{false}, \perp\}$.

A common technique to reason about specifications with a three-valued semantics is to *eliminate* ill-defined terms before starting the actual proof. *Well-definedness conditions* are generated, whose validity ensures that all formulas at hand can be evaluated to either **true** or **false**. That is, once the well-definedness conditions have been discharged, $\perp$ is guaranteed to never be encountered.

The advantage of the technique is that both the well-definedness conditions and the actual proof obligations are to be proven in classical two-valued logic, which is simpler, better understood, more widely used, and has better automated tool support [30] than three-valued logics.

The technique of eliminating ill-defined terms in specifications by generating well-definedness conditions is used in several approaches, for instance, B [2], PVS [13], CVC Lite [6], and ESC/Java2 [21].

Motivation. A drawback of this approach is that well-definedness conditions can be very large, causing significant time overhead in the proof process. As an example, consider the following formula:

$$x \,/\, y = c_1 \;\wedge\; \mathit{fact}(y) = c_2 \;\wedge\; y > 5 \tag{1}$$

where x and y are variables, and c_1 and c_2 are constants. The formula is well-defined, that is, it always evaluates to either **true** or **false**. This can be justified by a case split on the third conjunct, which is always well-defined:

1. if the third conjunct evaluates to **true**, then the division and factorial functions are known to be applied within their domains, and thus, the first and second conjuncts can be evaluated to **true** or **false**. This means that the whole formula can be evaluated to either **true** or **false**.
2. if the third conjunct evaluates to **false**, then the whole formula evaluates to **false** (according to the semantics we use in the paper).

The literature [8, 27, 3, 9] proposes the procedure $\mathcal{D}$ to generate well-definedness conditions. The procedure is *complete* [8, 9], that is, the well-definedness condition generated from a formula is provable if and only if the formula is well-defined. Procedure $\mathcal{D}$ would generate the following condition for (1):

$$\begin{array}{l}(y \neq 0 \wedge (y \geq 0 \vee (y \geq 0 \wedge \mathit{fact}(y) \neq c_2) \vee y \leq 5)) \;\vee \\ (y \neq 0 \wedge x/y \neq c_1) \;\vee \\ ((y \geq 0 \vee (y \geq 0 \wedge \mathit{fact}(y) \neq c_2) \vee y \leq 5) \wedge \neg(\mathit{fact}(y) \neq c_2 \wedge y > 5))\end{array}$$

As expected, the condition is provable. However, the size of the condition is striking, given that the original formula contained only three sub-formulas and two partial-function applications. In fact, procedure $\mathcal{D}$ yields well-formedness conditions that *grow exponentially* with respect to the input formula. This is a major problem for tools that have to prove well-definedness of considerably larger formulas than (1), for instance, the well-definedness conditions for B models, as presented in [8].

Due to the exponential blow-up of well-definedness conditions, the $\mathcal{D}$ procedure is not used in practice [8, 27, 3]. Instead, another procedure $\mathcal{L}$ is used, which generates much smaller conditions with linear growth, but which is *incomplete*. That is, the procedure may generate unprovable well-definedness conditions for well-defined formulas. This is the case with formula (1), for which the procedure would yield the following unprovable condition:

$$y \neq 0 \wedge (x/y = c_1 \Rightarrow y \geq 0)$$

Incompleteness of the procedure originates from its "sensitivity" to the order of sub-formulas. For instance, after proper re-ordering of the sub-formulas of (1), the procedure would yield a provable condition. This may be tedious for large formulas and may appear unnatural to users who are familiar with logics in which the order of sub-formulas is irrelevant for proof. Furthermore, there are situations (for instance, our example in Section 4) where such a manual re-ordering cannot be done.

Contributions. Our main contribution is a new procedure $\mathcal{Y}$, which unifies the advantages of $\mathcal{D}$ and $\mathcal{L}$, while eliminating their weaknesses. That is, (1) $\mathcal{Y}$ yields well-definedness conditions that *grow linearly* with respect to the size of the input formula, and (2) $\mathcal{Y}$ is equivalent to $\mathcal{D}$, and therefore complete and insensitive to the order of sub-formulas. To our knowledge, this is the first procedure that has *both* of these two properties.

The definition of the new procedure is very intuitive and straightforward. We prove that it is equivalent with $\mathcal{D}$ in two ways: (1) in a syntactical manner, as $\mathcal{D}$ was derived in [3], and (2) in a semantical way, as $\mathcal{D}$ was introduced in [8].

We have implemented the new procedure in the Spec# verification tool [5] in the context of the well-formedness checking of method specifications [26]. We have compared our procedure with $\mathcal{D}$ and $\mathcal{L}$ using two automated theorem provers. The empirical results clearly show that not only the size of generated well-definedness conditions are significantly smaller than what $\mathcal{D}$ produces, but the time to prove validity of the conditions is also decreased by the use of $\mathcal{Y}$. Furthermore, our results show that the performance of $\mathcal{Y}$ is also better than that of $\mathcal{L}$ in terms of the size of generated conditions.

Outline. The rest of the paper is structured as follows. Section 2 formally defines procedures $\mathcal{D}$ and $\mathcal{L}$, and highlights their main differences. Section 3 presents our main contribution: the $\mathcal{Y}$ procedure and the proof of its equivalence with $\mathcal{D}$. Section 4 demonstrates the improvements of our approach through empirical results. We discuss related work in Section 5 and conclude in Section 6.

2 Eliminating Ill-definedness

The main idea behind the technique of eliminating ill-definedness in specifications is to reduce the three-valued domain to a two-valued domain by ensuring that $\perp$ is never encountered. $\mathcal{D}$ is used for this purpose. Hoogewijs introduced $\mathcal{D}$ in the form of the logical connective Δ in [19], and proposed a first-order calculus, which includes this connective. Later, $\mathcal{D}$ was reformulated as a formula transformer, for instance, in [8, 3, 9] for the above syntax. $\mathcal{D}$ takes a formula ϕ and produces another formula $\mathcal{D}(\phi)$. The interpretation of the formula $\mathcal{D}(\phi)$ in two-valued logic is **true** if and only if the interpretation of ϕ in three-valued logic is different from $\perp$.

$$
\begin{array}{lll}
\textit{Term} ::= \textit{Var} & \qquad & \textit{Formula} ::= P(t_1,\ldots,t_n) \\
\quad\;\;\mid\; f(t_1,\ldots,f_n) & & \qquad\qquad\mid\; \textit{true} \quad\mid\; \textit{false} \\
& & \qquad\qquad\mid\; \neg\phi \\
& & \qquad\qquad\mid\; \phi_1 \wedge \phi_2 \;\mid\; \phi_1 \vee \phi_2 \\
& & \qquad\qquad\mid\; \forall\, x.\ \phi \;\;\mid\; \exists\, x.\ \phi
\end{array}
$$

Fig. 1. Syntax of terms and formulas we consider in this paper.

$$
\begin{array}{lll}
\delta(Var) & \triangleq & \textit{true} \\
\delta(f(e_1,\ldots,e_n)) & \triangleq & d_f(e_1,\ldots,e_n) \;\wedge\; \bigwedge_{i=1}^{n} \delta(e_i) \\
\mathcal{D}(P(e_1,\ldots,e_n)) & \triangleq & \bigwedge_{i=1}^{n} \delta(e_i) \\
\mathcal{D}(\textit{true}) & \triangleq & \textit{true} \\
\mathcal{D}(\textit{false}) & \triangleq & \textit{true} \\
\mathcal{D}(\neg\phi) & \triangleq & \mathcal{D}(\phi) \\
\mathcal{D}(\phi_1 \wedge \phi_2) & \triangleq & (\mathcal{D}(\phi_1) \wedge \mathcal{D}(\phi_2)) \;\vee\; (\mathcal{D}(\phi_1) \wedge \neg\phi_1) \;\vee\; (\mathcal{D}(\phi_2) \wedge \neg\phi_2) \\
\mathcal{D}(\phi_1 \vee \phi_2) & \triangleq & (\mathcal{D}(\phi_1) \wedge \mathcal{D}(\phi_2)) \;\vee\; (\mathcal{D}(\phi_1) \wedge \phi_1) \;\vee\; (\mathcal{D}(\phi_2) \wedge \phi_2) \\
\mathcal{D}(\forall\, x.\ \phi) & \triangleq & \forall\, x.\, \mathcal{D}(\phi) \;\vee\; \exists\, x.\, (\mathcal{D}(\phi) \wedge \neg\phi) \\
\mathcal{D}(\exists\, x.\ \phi) & \triangleq & \forall\, x.\, \mathcal{D}(\phi) \;\vee\; \exists\, x.\, (\mathcal{D}(\phi) \wedge \phi)
\end{array}
$$

Fig. 2. Definition of the δ and $\mathcal{D}$ operators as given by Behm et al. [8].

In order to have a basis for formal definitions, we define the syntax of terms and formulas that we consider in this paper. We follow the standard syntax-definition given in Figure 1. Throughout the paper we use **true**, **false**, and $\perp$ to denote the semantic truth values, and *true* and *false* to refer to the syntactic entities.

2.1 Defining the $\mathcal{D}$ Operator

The definition of $\mathcal{D}$ is given in Figure 2. Operator δ handles terms and $\mathcal{D}$ handles formulas. A variable is always well-defined. Application of function f is well-defined if and only if f's *domain restriction* d_f holds and all parameters e_i are well-defined. Each function is associated with a domain restriction, which is a predicate that represents the domain of the function. Such predicates should only contain total-function applications. For instance, the domain restriction of the factorial function is that the parameter is non-negative.

A predicate is well-defined if and only if all parameters are well-defined. Note that this definition assumes predicates to be total. Although an extension to partial predicates is straightforward, we use this definition for simplicity and to have a direct comparison of our approach to [8]. Constants *true* and *false* are always well-defined. Well-definedness of logical connectives is defined according to Strong Kleene connectives [22]. For instance, as the truth table in Figure 3(a)

$\wedge$	**true**	**false**	$\perp$
true	**true**	**false**	$\perp$
false	**false**	**false**	**false**
$\perp$	$\perp$	**false**	$\perp$

(a) Strong Kleene

$\wedge$	**true**	**false**	$\perp$
true	**true**	**false**	$\perp$
false	**false**	**false**	**false**
$\perp$	$\perp$	$\perp$	$\perp$

(b) McCarthy

Fig. 3. Kleene's and McCarthy's interpretation of conjunction.

shows, a conjunction is well-defined if and only if either (1) both conjuncts are well-defined, or (2) if one of the conjuncts is well-defined and evaluates to **false**. Intuitively, in case (1) the classical two-valued evaluation can be applied, while in case (2) the truth value of the conjunction is **false** independently of the well-definedness and value of the other conjunct.

Well-definedness of universal quantification can be thought of as the generalization of the well-definedness of conjunction. Disjunction and existential quantification are the duals of conjunction and universal quantification, respectively. Soundness and completeness of $\mathcal{D}$ was proven in [19, 8, 9].

2.2 An Approximation of the $\mathcal{D}$ Operator

As mentioned before in Section 1, the problem with the $\mathcal{D}$ operator is that it yields well-definedness conditions that grow exponentially with respect to the size of the input formula. This problem has been recognized in several approaches, for instance, in B [3] and PVS [27]. As a consequence, these approaches use a simpler, but stricter operator $\mathcal{L}$ [8, 3] for computing well-definedness conditions. The definition of $\mathcal{L}$ differs from that of $\mathcal{D}$ only for the following connectives:[1]

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\mathcal{L}(\phi_1 \wedge \phi_2) & \triangleq & \mathcal{L}(\phi_1) \wedge (\phi_1 \Rightarrow \mathcal{L}(\phi_2)) & \mathcal{L}(\forall\, x.\ \phi) & \triangleq & \forall\, x.\, \mathcal{L}(\phi) \\
\mathcal{L}(\phi_1 \vee \phi_2) & \triangleq & \mathcal{L}(\phi_1) \wedge (\neg\phi_1 \Rightarrow \mathcal{L}(\phi_2)) & \mathcal{L}(\exists\, x.\ \phi) & \triangleq & \forall\, x.\, \mathcal{L}(\phi)
\end{array}$$

Looking at the definition, we can see that $\mathcal{L}$ yields well-definedness conditions that grow linearly with respect to the input formula. This is a great advantage over $\mathcal{D}$. However, the $\mathcal{L}$ operator is stronger than $\mathcal{D}$, that is, $\mathcal{L}(\phi) \Rightarrow \mathcal{D}(\phi)$ holds, but $\mathcal{D}(\phi) \Rightarrow \mathcal{L}(\phi)$ does not necessarily hold, as shown for formula (1) in Section 1. This means that we lose completeness with the use of $\mathcal{L}$.

For quantifiers, $\mathcal{L}$ requires that the quantified formula is well-defined for all instantiations of the quantified variable. As a result, a universal quantification may be considered ill-defined although an instance is known to evaluate to **false**. Similarly, an existential quantification may also be considered ill-defined although an instance is known to evaluate to **true**. The $\mathcal{D}$ operator takes these "short-circuits" into account.

[1] Although our formula-syntax does not contain implication, we use it below to keep the intuition behind the definition.

The other source of incompleteness originates from defining conjunction and disjunction according to McCarthy's interpretation [24], which evaluates formulas *sequentially*. That is, if the first operand of a connective is $\perp$, then the result is defined to be $\perp$, independently of the second operand. The truth table in Figure 3(b) presents McCarthy's interpretation of conjunction. The only difference from Kleene's interpretation is in the interpretation of $\perp \wedge$ **false**, which yields $\perp$. This reveals the most important difference between the two interpretations: in McCarthy's interpretation conjunction and disjunction are not commutative.

As a consequence, for instance, $\phi_1 \ \wedge \ \phi_2$ may be considered ill-defined, although $\phi_2 \ \wedge \ \phi_1$ is considered well-defined. Such cases might come unexpected to users who are used to classical logic where conjunction and disjunction are commutative.

In most cases this incompleteness issue can be resolved by manually re-ordering sub-formulas. However, as pointed out by Rushby et al. [27], the manual re-ordering of sub-formulas is not an option when specifications are automatically generated from some other representation. Furthermore, Cheng and Jones [12], and Rushby et al. [27] give examples for which even manual re-ordering does not help, and well-defined formulas are inevitably rejected by $\mathcal{L}$.

3 An Efficient Equivalent of the $\mathcal{D}$ Operator

In this section we present our main contribution: a new procedure $\mathcal{Y}$ that yields considerably smaller well-definedness conditions than $\mathcal{D}$, and that retains completeness. We prove equivalence of $\mathcal{Y}$ and $\mathcal{D}$ in two ways: (1) we syntactically derive the definition of $\mathcal{Y}$, (2) using three-valued interpretation we prove by induction that the definition of $\mathcal{Y}$ is equivalent to that of $\mathcal{D}$. Both proofs demonstrate the intuitive and simple nature of $\mathcal{Y}$'s definition.

3.1 Syntactical Derivation of $\mathcal{Y}$

We introduce two new formula transformers $\mathcal{T}$ and $\mathcal{F}$, and define them as follows:

$$\mathcal{T}(\phi) \ \triangleq \ \mathcal{D}(\phi) \ \wedge \ \phi \qquad \text{and} \qquad \mathcal{F}(\phi) \ \triangleq \ \mathcal{D}(\phi) \ \wedge \ \neg\,\phi$$

That is, $\mathcal{T}(\phi)$ yields **true** if and only if ϕ is well-defined and evaluates to **true**. Analogously, $\mathcal{F}(\phi)$ yields **true** if and only if ϕ is well-defined and evaluates to **false**. From the definitions the following theorem follows.

Theorem 1. $\mathcal{D}(\phi) \ \Leftrightarrow \ \mathcal{T}(\phi) \ \vee \ \mathcal{F}(\phi)$
Proof. $\mathcal{D}(\phi) \ \Leftrightarrow \ \mathcal{D}(\phi) \ \wedge \ (\phi \ \vee \ \neg\,\phi) \ \Leftrightarrow \ (\mathcal{D}(\phi) \ \wedge \ \phi) \ \vee \ (\mathcal{D}(\phi) \ \wedge \ \neg\,\phi) \ \Leftrightarrow$ $\mathcal{T}(\phi) \ \vee \ \mathcal{F}(\phi)$ □

Intuitively, the theorem expresses that formula ϕ is well-defined if and only if ϕ evaluates either to **true** or to **false**. This directly corresponds to the interpretation of $\mathcal{D}$ given by Hoogewijs [19].

From the definitions of $\mathcal{T}$ and $\mathcal{F}$, the equivalences presented in Figure 4 can be derived. To demonstrate the simplicity of these derivations, we give the derivation of $\mathcal{T}(\phi_1 \wedge \phi_2)$ and $\mathcal{F}(\forall x.\ \phi)$:

$$\begin{array}{l}
\mathcal{T}(\phi_1 \wedge \phi_2) \;\Leftrightarrow\; \mathcal{D}(\phi_1 \wedge \phi_2) \wedge \phi_1 \wedge \phi_2 \;\Leftrightarrow \\
((\mathcal{D}(\phi_1) \wedge \mathcal{D}(\phi_2)) \vee (\mathcal{D}(\phi_1) \wedge \neg\phi_1) \vee (\mathcal{D}(\phi_2) \wedge \neg\phi_2)) \wedge \phi_1 \wedge \phi_2 \;\Leftrightarrow \\
\mathcal{D}(\phi_1) \wedge \mathcal{D}(\phi_2) \wedge \phi_1 \wedge \phi_2 \;\Leftrightarrow\; \mathcal{T}(\phi_1) \wedge \mathcal{T}(\phi_2)
\end{array}$$

$$\begin{array}{l}
\mathcal{F}(\forall x.\ \phi) \;\Leftrightarrow\; \mathcal{D}(\forall x.\ \phi) \wedge \neg\forall x.\ \phi \;\Leftrightarrow \\
(\forall x.\ \mathcal{D}(\phi) \vee (\exists x.\ (\mathcal{D}(\phi) \wedge \neg\phi))) \wedge \exists x.\ \neg\phi \;\Leftrightarrow \\
\exists x.\ (\mathcal{D}(\phi) \wedge \neg\phi) \;\Leftrightarrow\; \exists x.\ \mathcal{F}(\phi)
\end{array}$$

The derived equivalences are very intuitive. Both $\mathcal{T}$ and $\mathcal{F}$ reflect the standard two-valued interpretation of formulas. For instance, $\mathcal{F}$ essentially realizes de Morgan's laws. The handling of terms is the same as before using the δ operator. Note that $\mathcal{T}$ and $\mathcal{F}$ are mutually recursive in the equivalences. Termination of the mutual application of the operators is trivially guaranteed: the size of formulas yields the measure for termination.

The more involved semantic proof of equivalence is presented in the appendix. The proof, in particular **Lemma 4**, highlights the intuition behind $\mathcal{Y}$'s definition.

The definition of our new procedure $\mathcal{Y}$, based on **Theorem 1**, is the following:

$$\mathcal{Y}(\phi) \;\triangleq\; \mathcal{T}(\phi) \;\vee\; \mathcal{F}(\phi)$$

It is easy to see that (1) our procedure begins by duplicating the size of the input formula ϕ, and (2) afterwards applies operators $\mathcal{T}$ and $\mathcal{F}$ that yield formulas that are linear in size with respect to their input formulas.

That is, overall our procedure yields well-definedness conditions that grow linearly with respect to the size of the input formula. This is a significant improvement over $\mathcal{D}$ which yields formulas that are exponential in size with respect to the input formula. Intuitively, this improvement can be explained as follows: $\mathcal{D}$ makes case distinctions on the well-definedness of sub-formulas at each step of its application, whereas $\mathcal{Y}$ only performs a single initial case distinction on the validity of the entire formula. In spite of this difference, our procedure is equivalent to $\mathcal{D}$, thus it is symmetric, as opposed to $\mathcal{L}$.

4 Implementation and Empirical Results

We have implemented a well-formedness checker in the context of the verification of object-oriented programs. Our implementation extends the Spec# verification tool [5] by a new module that performs well-definedness and well-foundedness checks on specifications using the $\mathcal{Y}$ procedure. Details of the technique applied in the well-formedness checker are described in [26].

$$
\begin{array}{lcllcl}
\mathcal{T}(P(e_1,..,e_n)) & \Leftrightarrow & P(e_1,..,e_n) \wedge \bigwedge_{i=1}^{n} \delta(e_i) & \mathcal{F}(P(e_1,..,e_n)) & \Leftrightarrow & \neg P(e_1,..,e_n) \wedge \bigwedge_{i=1}^{n} \delta(e_i) \\
\mathcal{T}(\mathit{true}) & \Leftrightarrow & \mathit{true} & \mathcal{F}(\mathit{true}) & \Leftrightarrow & \mathit{false} \\
\mathcal{T}(\mathit{false}) & \Leftrightarrow & \mathit{false} & \mathcal{F}(\mathit{false}) & \Leftrightarrow & \mathit{true} \\
\mathcal{T}(\neg\phi) & \Leftrightarrow & \mathcal{F}(\phi) & \mathcal{F}(\neg\phi) & \Leftrightarrow & \mathcal{T}(\phi) \\
\mathcal{T}(\phi_1 \wedge \phi_2) & \Leftrightarrow & \mathcal{T}(\phi_1) \wedge \mathcal{T}(\phi_2) & \mathcal{F}(\phi_1 \wedge \phi_2) & \Leftrightarrow & \mathcal{F}(\phi_1) \vee \mathcal{F}(\phi_2) \\
\mathcal{T}(\phi_1 \vee \phi_2) & \Leftrightarrow & \mathcal{T}(\phi_1) \vee \mathcal{T}(\phi_2) & \mathcal{F}(\phi_1 \vee \phi_2) & \Leftrightarrow & \mathcal{F}(\phi_1) \wedge \mathcal{F}(\phi_2) \\
\mathcal{T}(\forall x.\ \phi) & \Leftrightarrow & \forall x.\ \mathcal{T}(\phi) & \mathcal{F}(\forall x.\ \phi) & \Leftrightarrow & \exists x.\ \mathcal{F}(\phi) \\
\mathcal{T}(\exists x.\ \phi) & \Leftrightarrow & \exists x.\ \mathcal{T}(\phi) & \mathcal{F}(\exists x.\ \phi) & \Leftrightarrow & \forall x.\ \mathcal{F}(\phi)
\end{array}
$$

Fig. 4. Derived equivalences for $\mathcal{T}$ and $\mathcal{F}$.

Additionally, in order to be able to compare the different procedures, we have built a prototype that implements the $\mathcal{D}$, $\mathcal{L}$, and $\mathcal{Y}$ procedures for the syntax given in Figure 1. We used the two automated theorem provers that are integrated with Spec#: Simplify [15] and Z3 [14], both of which are used by several other tools as prover back-ends too. The experiment was performed on a machine with Intel Pentium M (1.86 GHz) and 2 GB RAM.

The benchmark. We have used the following inductively defined formula, which allowed us to experiment with formula sizes that grow linearly with respect to n, and which is well-defined for every natural number n:

$$
\begin{array}{lcl}
\phi_0 & \triangleq & f(x) = x \ \vee \ f(-x) = -x \\
\phi_n & \triangleq & \phi_{n-1} \ \wedge \ (f(x+n) = x+n \ \vee \ f(-x-n) = -x-n)
\end{array}
$$

where the definition and domain restriction of f is as follows:

$$\forall x.\ x \geq 0 \ \Rightarrow \ f(x) = x \qquad \text{and} \qquad d_f\!:\ x \geq 0$$

Note that formula ϕ_n is well-defined for any n. However, its well-definedness cannot be proven using $\mathcal{L}$ for any n, and no re-ordering would help this situation.

Empirical results. Figure 5(a) shows that well-definedness conditions generated by $\mathcal{D}$ grow exponentially, whereas conditions generated by $\mathcal{L}$ and $\mathcal{Y}$ grow linearly. This was expected from their definitions. Note that the y axis uses a logarithmic scale. The figure also shows, that the sizes of conditions generated using $\mathcal{Y}$ are smaller than those generated by $\mathcal{L}$ for $n > 4$.

Figure 5(b) compares the time that Simplify (version 1.5.4) required to prove the well-definedness conditions generated from our input formula. As required by its interface, these conditions were given to Simplify as plain text. We see that the time required to prove formulas generated by $\mathcal{D}$ grows exponentially, whereas with $\mathcal{Y}$ the required time grows linearly. Note that the y axis uses a logarithmic scale. Additionally, for $\mathcal{D}$ our prototype was not able to generate the well-definedness condition for input formulas with $n > 16$ because it ran out of memory.

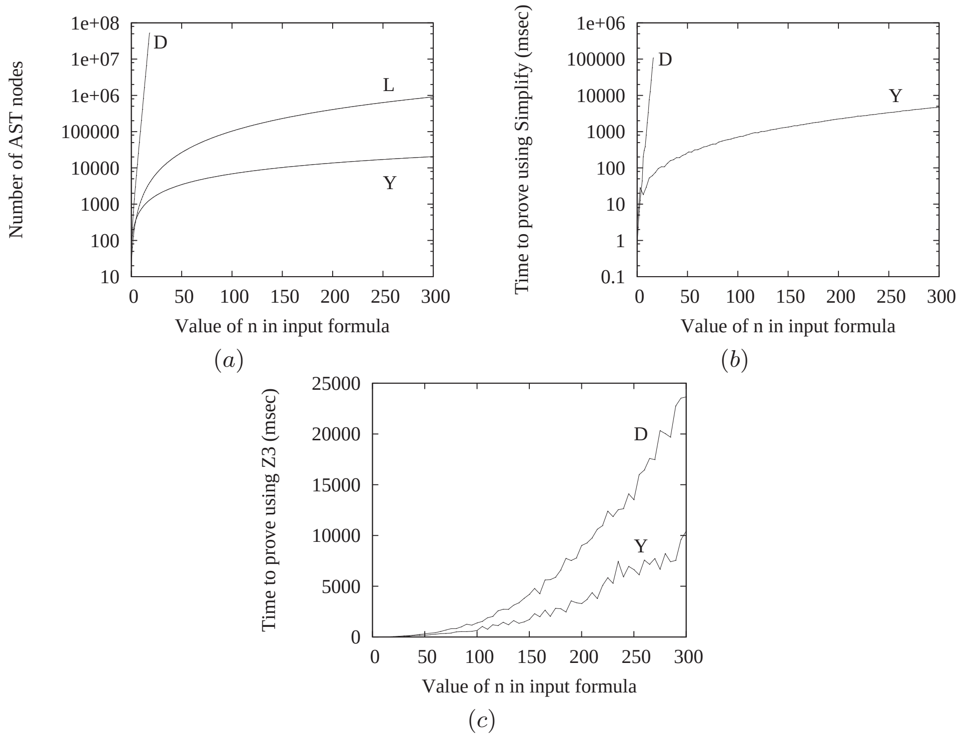


Fig. 5. (a) Size of well-definedness conditions generated by procedures $\mathcal{D}$, $\mathcal{L}$, and $\mathcal{Y}$; (b) Time to prove well-definedness conditions using Simplify; (c) Time to prove well-definedness conditions using Z3.

Figure 5(c) shows the results of the same experiment using Z3 (version 1.2). Note that the y axis is linear. From this graph we see that although the times required to prove well-definedness conditions show the same growth pattern for both procedure $\mathcal{D}$ and $\mathcal{Y}$, the times recorded for $\mathcal{Y}$ are approximately 1/3 to 1/2 below that of for $\mathcal{D}$. For instance, with $n = 200$, Z3 proves the condition generated by $\mathcal{D}$ in 9 seconds, while it takes 3.5 seconds for the condition generated by $\mathcal{Y}$. For $n = 300$, these figures are 23.5 and 10.5 seconds, respectively. Note that we could successfully prove much larger well-definedness conditions generated by $\mathcal{D}$ in Z3 as compared to Simplify. This is because (1) we used the native API of Z3 in order to construct formulas with maximal sharing, and (2) due to the use of its API, Z3 may have benefited from sub-formula sharing, which could have made the size of the resulting formula representation linear. In spite of this, procedure $\mathcal{Y}$ performs better than $\mathcal{D}$.

Note that Figure 5(b) and 5(c) do not plot the results of $\mathcal{L}$. This is because the $\mathcal{L}$ procedure cannot prove well-definedness of the input formulas.

Finally, we note that the whole sequence of formulas were passed to a single session of Simplify or Z3, respectively.

5 Related Work

The handling of partial functions in formal specifications has been studied extensively and several different approaches have been proposed. Here we only mention three mainstream approaches and refer the reader for detailed accounts to Arthan's paper [4], which classifies different approaches to undefinedness, to Abrial and Mussat's paper [3, Section 1.7], and to Hähnle's survey [18].

Eliminating undefinedness. As mentioned already in the paper, eliminating undefinedness by the generation of well-definedness conditions is a common technique to handle partial functions, and is applied in several approaches, such as B [8, 3], PVS [27], CVC Lite [9], and ESC/Java2 [11]. The two procedures proposed in these papers are $\mathcal{D}$ and $\mathcal{L}$.

PVS combines proving well-definedness conditions with type checking. In PVS, partial functions are modeled as total functions whose domain is a predicate subtype [27]. This makes the type system undecidable requiring Type Correctness Conditions to be proven. PVS uses the $\mathcal{L}$ operator because $\mathcal{D}$ was found to be inefficient [27].

CVC Lite uses the $\mathcal{D}$ procedure for the well-definedness checking of formulas. Berezin et al. [9] mention that if formulas are represented as DAGs, then the worst-case size of $\mathcal{D}(\phi)$ is linear with respect to the size of ϕ. However, there are no empirical results presented to confirm any advantages of using the DAG representation in terms of proving times.

Recent work on ESC/Java2 by Chalin [11] requires proving the well-definedness of specifications written in the Java Modeling Language (JML) [23]. Chalin uses the $\mathcal{L}$ procedure, however, as opposed to other approaches, not because of inefficiency issues. The $\mathcal{L}$ procedure directly captures the semantics of conditional boolean operators (e.g. `&&` and `||` in Java) that many programming languages contain, and which are often used, for instance, in JML specifications. Chalin's survey [10] indicates that the use of $\mathcal{L}$ is better suited for program verification than $\mathcal{D}$, since it yields well-definedness conditions that are closer to the expectations of programmers.

Schieder and Broy [28] propose a different approach to the checking of well-definedness of formulas. They define a formula under a three-valued interpretation to be well-defined if and only if its interpretation yields **true** both if $\perp$ is interpreted as **true**, and if $\perp$ is interpreted as **false**. Although checking well-definedness of formulas becomes relatively simple, the interpretation may be unintuitive for users. For example, formula $\perp \vee \neg\perp$ is considered to be well-defined. We prefer to eliminate such formulas by using classical Kleene logic.

Three-valued logics. Another standard way to handle partial functions is to fully integrate ill-defined terms into the formal logic by developing a three-valued logic. This approach is attributed to Kleene [22]. A well-known three-valued logic is LPF [7, 12] developed by C.B. Jones et al. in the context of VDM [20]. Other languages that follow this approach include Z [29] and OCL [1].

A well-known drawback of three-valued logics is that they may seem unnatural to proof engineers. For instance, in LPF, the law of the excluded middle and the deduction rule (a.k.a. ImpI) do not hold. Furthermore, a second notion of equality (called "weak equality") is required to avoid proving, for instance, that $x \,/\, 0 = \mathit{fact}(-5)$ holds. Another major drawback is that there is significantly less tool support for three-valued logics than there is for two-valued logics.

Underspecification. The approach of underspecification assigns an ill-defined term a definite, but unknown value from the type of the term [16]. Thus, the resulting interpretation is two-valued, however, in certain cases the truth value of formulas cannot be determined due to the unknown values. For instance, the truth value of $x \,/\, 0 = \mathit{fact}(-5)$ is known to be either **true** or **false**, but there is no way to deduce which of the two. However, for instance, $x \,/\, 0 = x \,/\, 0$ is trivially provable. This might not be a desired behavior. For instance, the survey by Chalin [10] argues that this is against the intuition of programmers, who would rather expect an error to occur in the above case. Underspecification is applied, for instance, in the Isabelle theorem prover [25], the Larch specification language [17], and JML [23].

6 Conclusion

A commonly applied technique to handle partial-function applications in formal specifications is to pose well-definedness conditions, which guarantee that undefined terms and formulas are never encountered. This technique allows one to use two-valued logic to reason about specifications that have a three-valued semantics. Previous work proposed two procedures, each having some drawback. The $\mathcal{D}$ procedure yields formulas that are too large to be used in practice. The $\mathcal{L}$ procedure is incomplete, resulting in the rejection of well-defined formulas.

In this paper we proposed a new procedure $\mathcal{Y}$, which eliminates these drawbacks: $\mathcal{Y}$ is complete and yields formulas that grow linearly with respect to the size of the input formula. Approaches that apply the $\mathcal{D}$ or $\mathcal{L}$ procedures (for instance, B, PVS, and CVC Lite) could benefit from our procedure. The required implementation overhead would be minimal.

Our procedure has been implemented in the Spec# verification tool to enforce well-formedness of invariants and method specifications. Additionally, we implemented a prototype to allow us to compare the new procedure with $\mathcal{D}$ and $\mathcal{L}$. Beyond the expected benefits of shorter well-definedness conditions, our experiments also show that theorem provers need less time to prove the conditions generated using $\mathcal{Y}$.

Acknowledgments. We would like to thank Peter Müller and the anonymous reviewers for their helpful comments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

1. UML 2.0 OCL Specification. Available at `http://www.omg.org/docs/formal/06-05-01.pdf`, May 2006.
2. J.-R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
3. J.-R. Abrial and L. Mussat. On using conditional definitions in formal theories. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002*, volume 2272 of *LNCS*, pages 242–269. Springer-Verlag, 2002.
4. R. Arthan. Undefinedness in Z: Issues for specification and proof. Presented at CADE Workshop on Mechanization of Partial Functions, 1996.
5. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.
6. C. W. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker category B. In R. Alur and D. Peled, editors, *CAV*, volume 3114 of *LNCS*, pages 515–518. Springer-Verlag, 2004.
7. H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.
8. P. Behm, L. Burdy, and J.-M. Meynadier. Well Defined B. In D. Bert, editor, *International B Conference*, volume 1393 of *LNCS*, pages 29–45. Springer-Verlag, 1998.
9. S. Berezin, C. Barrett, I. Shikanian, M. Chechik, A. Gurfinkel, and D. L. Dill. A practical approach to partial functions in CVC Lite. In *Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2004.
10. P. Chalin. Are the logical foundations of verifying compiler prototypes matching user expectations? *Formal Aspects of Computing*, 19(2):139–158, 2007.
11. P. Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *ICSE*, pages 23–33. IEEE Computer Society, 2007.
12. J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In *Refinement Workshop*, pages 51–69, 1991.
13. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, 1995.
14. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer-Verlag, 2008.
15. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.
16. D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In *Computer Science Today*, volume 1000 of *LNCS*, pages 366–373. Springer-Verlag, 1995.
17. J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.
18. R. Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IGPL*, 13(4):415–433, 2005.
19. A. Hoogewijs. On a formalization of the non-definedness notion. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 25:213–217, 1979.
20. C. B. Jones. *Systematic software development using VDM*. Prentice Hall, 1986.
21. J. R. Kiniry and D. R. Cok. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS*, volume 3362 of *LNCS*, pages 108–128. Springer-Verlag, 2005.

22. S. C. Kleene. On a notation for ordinal numbers. *Journal of Symbolic Logic*, 3:150–155, 1938.
23. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
24. J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.
25. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
26. A. Rudich, Á. Darvas, and P. Müller. Checking well-formedness of pure-method specifications. In J. Cuellar and T. Maibaum, editors, *Formal Methods (FM)*, volume 5014 of *LNCS*, pages 68–83. Springer-Verlag, 2008.
27. J. Rushby, S. Owre, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.
28. B. Schieder and M. Broy. Adapting calculational logic to the undefined. *The Computer Journal*, 42(2):73–81, 1999.
29. J. M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1988.
30. G. Sutcliffe, C. B. Suttner, and T. Yemenis. The TPTP Problem Library. In A. Bundy, editor, *CADE*, volume 814 of *LNCS*, pages 252–266. Springer-Verlag, 1994.

A Semantic Proof of Equivalence

Structures. We define structures and interpretations in a way similar to as Behm et al. [8]. Let $\mathbf{A}$ be a set that does not contain $\perp$. We define $\mathbf{A}_\perp$ as $\mathbf{A} \cup \{\perp\}$. Let $\mathbf{F}$ be a set of function symbols, and $\mathbf{P}$ a set of predicate symbols. Let $\mathbf{I}$ be a mapping from $\mathbf{F}$ to the set of functions from $\mathbf{A}^n$ to $\mathbf{A}_\perp$, and from $\mathbf{P}$ to the set of predicates from $\mathbf{A}^n$ to $\{\mathbf{true}, \mathbf{false}\}$ (for simplicity, we assume that the interpretation of predicates is total), where n is the arity of the corresponding function or predicate symbol. We say that $\mathbf{M} = \langle \mathbf{A}, \mathbf{I} \rangle$ is a structure for our language with carrier set $\mathbf{A}$ and interpretation $\mathbf{I}$. We call a structure total if the interpretation of every function $\mathbf{f} \in \mathbf{F}$ is total, which means $\mathbf{f}(\ldots) \neq \perp$. We call the structure partial otherwise. A partial structure $\mathbf{M}$ can be extended to a total structure $\hat{\mathbf{M}}$ by having functions evaluated outside their domains return arbitrary values.

Interpretation. For a term $\mathbf{t}$, structure $\mathbf{M}$, and variable assignment $\mathbf{e}$, we denote the interpretation of $\mathbf{t}$ as $[\mathbf{t}]^{\mathbf{e}}_{\mathbf{M}}$. *Variable assignment* $\mathbf{e}$ maps the free variables of $\mathbf{t}$ to values. We define the interpretation of terms as given in Figure 6. Interpretation of formula φ denoted as $[\varphi]^{\mathbf{e}}_{\mathbf{M}}$ is given in Figure 7. **Dom** yields the domain of the interpretation of function symbols.
To check whether or not a value $\mathbf{l}$ is defined, we use function $\mathbf{wd}$:

$$\mathbf{wd}(\mathbf{l}) = \begin{cases} \mathbf{true}, & \text{if } \mathbf{l} \in \{\mathbf{true}, \mathbf{false}\} \\ \mathbf{false}, & \text{if } \mathbf{l} = \perp \end{cases}$$

$$
\begin{array}{lcl}
[\,\mathbf{v}\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \mathbf{e}(\mathbf{v}) \text{ where } \mathbf{v} \text{ is a variable} \\
[\,\mathbf{f}(\mathbf{t}_1,\ldots,\mathbf{t}_n)\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{I}(\mathbf{f})([\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}},\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}), \text{ if } \langle[\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}},\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}\rangle \in \mathbf{Dom}(\mathbf{I}(\mathbf{f})) \\ \qquad\qquad\qquad\qquad \text{and } [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}} \neq \bot,\ldots,\ [\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \neq \bot \\ \bot, \text{otherwise} \end{cases}
\end{array}
$$

Fig. 6. Interpretation of terms.

Lemma 1. *For every total structure* $\mathbf{M}$*, formula* φ*, and variable assignment* $\mathbf{e}$*, we have* $\mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true}$*.*
Proof. By induction over the structure of φ. □

Lemma 2. *For every structure* $\mathbf{M}$*, if* $\mathbf{M}$ *is extended to total structure* $\hat{\mathbf{M}}$*, then* $\mathbf{wd}([\varphi]^{\mathbf{e}}_{\hat{\mathbf{M}}}) = \mathbf{true}$*.*
Proof. Trivial consequence of the way $\mathbf{M}$ is extended and of **Lemma 1**. □

Domain restrictions. Each function $\mathbf{f}$ is associated with a domain restriction d_f, which is a predicate that represents the domain of function $\mathbf{f}$. A structure $\mathbf{M}$ is a model for domain restrictions of functions in $\mathbf{F}$ (denoted by $d_{\mathbf{F}}(\mathbf{M})$) if and only if:

– The domain formulas are defined. That is, for each $\mathbf{f} \in \mathbf{F}$ and for all $\mathbf{e}$:
  $\mathbf{wd}([d_f]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true}$
– Domain restrictions characterize the domains of function interpretations for $\mathbf{M}$. That is, for each $\mathbf{f} \in \mathbf{F}$ and $\mathbf{l}_1,\ldots,\mathbf{l}_n \in \mathbf{A}$:
  $[d_f]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ if and only if $\langle \mathbf{l}_1,\ldots,\mathbf{l}_n\rangle \in \mathbf{Dom}(\mathbf{I}(f))$
  where $e = [v_1 \to \mathbf{l}_1,\ldots,v_n \to \mathbf{l}_n]$ and $\{v_1,\ldots,v_k\}$ are $\mathbf{f}$'s parameter names.

In the following we prove two lemmas and finally our two main theorems.

Lemma 3. *For each structure* $\mathbf{M}$*, term* $\mathbf{t}$*, and variable assignment* $\mathbf{e}$*:*
if $d_{\mathbf{F}}(\mathbf{M})$ *then* $[\mathbf{t}]^{\mathbf{e}}_{\mathbf{M}} \neq \bot$ *if and only if* $[\delta(\mathbf{t})]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$*.*
Proof. By induction on the structure of $\mathbf{t}$ under the assumption that $d_{\mathbf{F}}(\mathbf{M})$.
Induction base: $\mathbf{t}$ is variable $\mathbf{v}$.
Since $[\mathbf{v}]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{e}(\mathbf{v}) \neq \bot$ and $[\delta(\mathbf{v})]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$, we have the desired property.
Induction step: $\mathbf{t}$ is function application $\mathbf{f}(\mathbf{t}_1,\ldots,\mathbf{t}_n)$.
From definition of interpretation we get that $[\mathbf{f}(\mathbf{t}_1,\ldots,\mathbf{t}_n)]^{\mathbf{e}}_{\mathbf{M}} \neq \bot$ if and only if:

$$\langle[\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}},\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}\rangle \in \mathbf{Dom}(\mathbf{I}(\mathbf{f})) \ \wedge\ [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}} \neq \bot \wedge \ldots \wedge [\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \neq \bot$$

By the definition of $d_{\mathbf{F}}(\mathbf{M})$ and the induction hypothesis, it is equivalent to:

$$[d_f(\mathbf{t}_1,\ldots,\mathbf{t}_n)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true} \wedge [\delta(\mathbf{t}_1)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true} \wedge \ldots \wedge [\delta(\mathbf{t}_n)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$$

$$
\begin{array}{lll}
[\,\mathit{true}\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \mathbf{true} \\
[\,\mathit{false}\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \mathbf{false} \\
[\,P(\mathbf{t}_1,\ldots,\mathbf{t}_n)\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if } \mathbf{I}(P)([\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}},\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true} \text{ and} \\ & \quad [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}} \neq \bot,\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \neq \bot \\ \mathbf{false}, & \text{if } \mathbf{I}(P)([\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}},\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{false} \text{ and} \\ & \quad [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}} \neq \bot,\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \neq \bot \\ \bot, & \text{otherwise} \end{cases} \\
[\,\neg\varphi\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false} \\ \mathbf{false}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\varphi \wedge \phi\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \text{ and } [\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \\ \mathbf{false}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false} \text{ or } [\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\varphi \vee \phi\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \text{ or } [\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \\ \mathbf{false}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false} \text{ and } [\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\forall\, x.\ \varphi\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if for all } \mathbf{l} \in \mathbf{A},\ [\varphi]^{\mathbf{e}[x\leftarrow\mathbf{l}]}_{\mathbf{M}} = \mathbf{true} \\ \mathbf{false}, & \text{if there exists } \mathbf{l} \in \mathbf{A} \text{ such that} [\varphi]^{\mathbf{e}[x\leftarrow\mathbf{l}]}_{\mathbf{M}} = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\exists\, x.\ \varphi\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if there exists } \mathbf{l} \in \mathbf{A} \text{ such that} [\varphi]^{\mathbf{e}[x\leftarrow\mathbf{l}]}_{\mathbf{M}} = \mathbf{true} \\ \mathbf{false}, & \text{if for all } \mathbf{l} \in \mathbf{A},\ [\varphi]^{\mathbf{e}[x\leftarrow\mathbf{l}]}_{\mathbf{M}} = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 7. Interpretation of formulas.

which is, by the definition of δ, equivalent to $[\delta(\mathbf{f}(\mathbf{t}_1,\ldots,\mathbf{t}_n))]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$. □

Lemma 4. *For each structure* $\mathbf{M}$*, formula* φ*, and variable assignment* $\mathbf{e}$*:*
if $d_{\mathbf{F}}(\mathbf{M})$ *then* $[\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ *if and only if* $[\mathcal{T}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$ *and*
$[\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false}$ *if and only if* $[\mathcal{F}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$*.*

Proof. By induction on the structure of φ under the assumption that $d_{\mathbf{F}}(\mathbf{M})$.
Induction base: φ is predicate $P(\mathbf{t}_1,\ldots,\mathbf{t}_n)$.
From definition of interpretation we get $[P(\mathbf{t}_1,\ldots,\mathbf{t}_n)]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ if and only if:

$$\mathbf{I}(P)([\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}},\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true} \ \wedge\ [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}} \neq \bot \wedge \ldots \wedge [\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \neq \bot$$

which is, by the assumption that the interpretation of predicates is total and by **Lemma 3**, equivalent to:

$$[P(\mathbf{t}_1,\ldots,\mathbf{t}_n)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true} \wedge [\delta(\mathbf{t}_1)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true} \wedge \ldots \wedge [\delta(\mathbf{t}_n)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$$

which is, by the definition of $\mathcal{T}$, equivalent to $[\mathcal{T}(P(\mathbf{t}_1,\ldots,\mathbf{t}_n))]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$.
The proof is analogous for $\mathcal{F}$.
Induction step: For brevity, we only present the proof of those two cases for which the syntactic derivation was shown on page 7. The proofs are analogous for all other cases.

1. We prove that if $d_{\mathbf{F}}(\mathbf{M})$ then $[\gamma\wedge\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ if and only if $[\mathcal{T}(\gamma\wedge\phi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$. From definition of interpretation we get that $[\gamma \wedge \phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ if and only if $[\gamma]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ and $[\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$, which is, by the induction hypothesis, equivalent to $[\mathcal{T}(\gamma)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$ and $[\mathcal{T}(\phi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$, which is, by the definition of $\mathcal{T}$, equivalent to $[\mathcal{T}(\gamma \wedge \phi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$.

2. We prove that if $d_{\mathbf{F}}(\mathbf{M})$ then $[\forall x.\ \phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false}$ iff $[\mathcal{F}(\forall x.\ \phi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$. From the definition of the interpretation function we get that $[\forall x.\ \phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false}$ if and only if there exists $\mathbf{l} \in \mathbf{A}$ such that $[\phi]^{\mathbf{e}[x\leftarrow\mathbf{l}]}_{\mathbf{M}} = \mathbf{false}$. By the induction hypothesis, this is equivalent to the existence of $\mathbf{l} \in \mathbf{A}$ such that $[\mathcal{F}(\phi)]^{\mathbf{e}[x\leftarrow\mathbf{l}]}_{\hat{\mathbf{M}}} = \mathbf{true}$, which is, by the definition of $\mathcal{F}$, equivalent to $[\mathcal{F}(\forall x.\ \phi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$. □

Theorem 2. *For each structure* $\mathbf{M}$*, formula* φ*, and variable assignment* $\mathbf{e}$*:*
if $d_{\mathbf{F}}(\mathbf{M})$ *then* $\mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}}) = [\mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}}$
Proof. From the definition of **wd** we know that $\mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}})$ is defined. Furthermore, from **Lemma 2** (with φ substituted by $\mathcal{Y}(\varphi)$) we know that $[\mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}}$ is defined. Thus, it is enough to prove that $\mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true}$ if and only if $[\mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$. Under the assumption that $d_{\mathbf{F}}(\mathbf{M})$, we have:
$\mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true}$ if and only if [by definition of **wd**]
$[\varphi]^{\mathbf{e}}_{\mathbf{M}} \in \{\mathbf{true}, \mathbf{false}\}$ if and only if [by **Lemma 4**]
$[\mathcal{T}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$ or $[\mathcal{F}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$ if and only if [by definition of $\mathcal{Y}$]
$[\mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$ □

Berezin et al. [6] proved the following characteristic property of $\mathcal{D}$:

$$\text{if } d_{\mathbf{F}}(\mathbf{M}) \text{ then } \mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}}) = [\mathcal{D}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} \qquad (2)$$

Theorem 3. *For each total structure* $\mathbf{M}$*, formula* φ*, and variable assignment* $\mathbf{e}$*:*
$[\mathcal{D}(\varphi) \Leftrightarrow \mathcal{Y}(\varphi)]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$
Proof. For each total structure $\mathbf{M}$ there exists a partial structure $\mathbf{M}'$ such that $\mathbf{M} = \hat{\mathbf{M}}'$ and $d_{\mathbf{F}}(\mathbf{M}')$. We can build $\mathbf{M}'$ from $\mathbf{M}$ by restricting the domain of partial functions according to the domain restrictions.
By **Theorem 2** and (2) we get $[\mathcal{D}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}'} = \mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}'}) = [\mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}'}$. Which is equivalent to $[\mathcal{D}(\varphi) \Leftrightarrow \mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}'} = \mathbf{true}$. Since $\mathbf{M} = \hat{\mathbf{M}}'$ we get the desired property. □