Behavioral Interface Specification Languages

JOHN HATCLIFF, Kansas State University
GARY T. LEAVENS, University of Central Florida
K. RUSTAN M. LEINO, Microsoft Research
PETER MÜLLER, ETH Zurich
MATTHEW PARKINSON, Microsoft Research Cambridge

Behavioral interface specification languages provide formal code-level annotations, such as preconditions, postconditions, invariants, and assertions that allow programmers to express the intended behavior of program modules. Such specifications are useful for precisely documenting program behavior, for guiding implementation, and for facilitating agreement between teams of programmers in modular development of software. When used in conjunction with automated analysis and program verification tools, such specifications can support detection of common code vulnerabilities, capture of light-weight application-specific semantic properties, generation of test cases and test oracles, and full formal program verification. This article surveys behavioral interface specification languages with a focus toward automatic program verification and with a view towards aiding the Verified Software Initiative—a fifteen-year, cooperative, international project directed at the scientific challenges of large-scale software verification.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers, class invariants, formal methods, model checking, programming by contract, reliability, validation*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Assertions, invariants, pre- and post-conditions, specification techniques*

General Terms: Design, Documentation, Reliability, Verification

Additional Key Words and Phrases: Abstraction, assertion, behavioral subtyping, frame conditions, interface specification language, invariant, JML, postcondition, precondition, separation logic, Spec#, SPARK

ACM Reference Format:
Hatcliff, J., Leavens, G. T., Leino, K. R. M., Müller, P., and Parkinson, M. 2012. Behavioral interface specification languages. ACM Comput. Surv. 44, 3, Article 16 (June 2012), 58 pages.
DOI = 10.1145/2187671.2187678 http://doi.acm.org/10.1145/2187671.2187678

1. INTRODUCTION

A software *behavorial specification* is a precise description of the intended behavior of some computing system or its components. Behavioral specifications take a variety of

16

The work of J. Hatcliff was supported in part by the US National Science Foundation (NSF) under grant CRI-0454348, the US Air Force Office of Scientific Research (AFOSR), and Rockwell Collins. The work of G.T. Leavens was supported in part by the US National Science Foundation under grants CNS 08-08913, CCF 0916350, and CCF 0916715. The work of P. Müller 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. The work of M. Parkinson was supported in part by a RAEng/EPSRC research fellowship and EPSRC grant EP/F019394.
Author's addresses: J. Hatcliff, Kansas State University; G. T. Leavens, University of Central Florida; K. Rustan M. Leino, Microsoft Research; P. Müller, Eth Zurich; M. Parkinson, Microsoft Research Cambridge; email (corresponding author): peter.mueller@inf.ethz.ch.

DOI 10.1145/2187671.2187678 http://doi.acm.org/10.1145/2187671.2187678

forms, and different specification notations play distinct roles in describing and constraining behavior. For example, type declarations or preconditions indicate the values that a component is designed to process. Assertions impose constraints on variable values as a system passes through different execution states. Postconditions specify the functionality of a component by describing how its output values relate to its input values. State machines provide a high-level view of system states and transitions between those states. Sequence charts summarize how a component interacts with other components or its environment. Use cases capture how users of different roles may interact with the system.

Numerous aspects of modern software and the processes by which it is constructed and validated are leading to an increased use of specifications.

—Modern software typically makes extensive use of reuseable component frameworks, including graphical user interface (GUI) libraries (e.g., Swing [Walrath et al. 2004], SWT[1]), frameworks for Web services and business applications (e.g., Apache Struts[2] and CSLA.Net [Lhotka 2008]), etc. Specifications on the interfaces of these frameworks are useful for declaring *software contracts* [Liskov and Guttag 1986; Meyer 1992] that precisely document the functionality provided by framework services, as well as the specific calling conventions that clients of the framework are obligated to follow to ensure correct operation of the provided services. Such specifications are also important for maintenance, since code alone cannot reveal what contract the code was intended to fulfill.

—The scale of software systems is increasing rapidly, and it is not uncommon for systems to have well over a million lines of code. In such large-scale systems, functionality must be decomposed and development tasks distributed to many teams. In some cases, teams may be located in different organizations and different geographical areas. Development processes may vary, and both language and cultural differences may introduce ambiguities and hinder communications. Imprecision in interfaces and team coding assignments can prevent the system from being correctly assembled from its components and can lead to signficant cost overruns. Use cases and sequence charts [Alhir 1998] can highlight key interactions of sytem components and achieve consensus during design. Interface specifications [Guttag et al. 1993; Jones 1990; Meyer 1997; Wing 1987, 1990] play a key role in cleanly partitioning the software system, recording the intended behavior of software components, achieving an appropriate division of labor among teams, increasing precision in communication between teams, and assuring that desired system functionality can be achieved by composing individually validated components.

—The sheer scale of software systems is leading to their extended use, since they are more costly to replace. At the same time, the pace of technology innovation is quickening. Unfortunately, this often leads to technology refresh problems—it is often difficult to refresh the software to accomodate advances in execution platforms, implementation languages, frameworks, or libraries. Specifications can provide implementation-independent descriptions of system behavior. In effect, they can serve as abstractions that make development less dependent on implementation details—facilitating change of hardware components and implementation strategies.

—Safety-critical systems including avionics, medical devices, automotive control systems, nuclear power plants, as well as systems associated with critical infrastructure, increasingly rely on software to provide their functionality. Verification and

[1]SWT: The Standard Widget Toolkit `http://eclipse.org/swt/`.

[2]Apache Struts–a free open-source framework for creating Java Web applications. `http://struts.apache.org`.

validation is a key component of the development and certification of such systems. Especially useful in these contexts, but in non-critical contexts as well, specifications provide a canonical declaration of a systems intended functionality against which an implementation can be verified. Specifications can also guide test case construction. When debugging, one can use specifications to isolate faults and assign blame [Findler and Felleisen 2002; Liskov and Guttag 1986; Meyer 1992].

The benefits of specifications are amplified when they are written in a *formal specification language*—a mathematically precise notation for recording intended properties of software. In this survey, we will consider specification languages with both a formal syntax and semantics. Formalizing the syntax of the specification language enables specifications to be processed by software development tools and checked for well formedness. Formalizing the semantics helps make specifications unambiguous, less dependent on cultural norms[3], and thus less likely to be misunderstood. More importantly, formalizing the semantics (e.g., using mathematical logic) enables tools to provide automated reasoning about specifications and their relationship to associated code.

Formal specifications can be leveraged by tools throughout the entire software life cycle. At design time, using automated deduction and SAT-based techniques, specifications can be checked automatically for consistency and queried to determine if desired system behaviors are implied by the specifications [Jackson 2006]. As coding begins, static analysis tools can check implementations against specifications, for example, a method body can be checked to determine if it correctly implements its contract, that it satisfies the preconditions of any methods that it calls, and that all assertions in the method body hold [Flanagan et al. 2002; Barnett et al. 2005]. Specifications can also serve as a starting place for transformational development in which specifications are systematically refined into code [Abadi and Lamport 1988; Abrial 1996; Hehner 1993; Jones 1990; Morgan 1994; Morgan and Vickers 1994; Morris 1980; Partsch and Steinbrüggen 1983]. Formal specifications need not specify full correctness to be useful; light-weight annotations including those that restrict ranges of numeric variables and that specify the non-nullness of reference variables can be easily incorporated during development to guide tools that seek to find bugs used deductive techniques [Flanagan et al. 2002; Barnett et al. 2005] or abstract interpretation [Blanchet et al. 2002]. During testing, executable representations of assertions can be checked, test cases can be generated automatically from formal specifications [Cartwright 1981; Chang and Richardson 1999; Boyapati et al. 2002], or implementations can be exercised directly from specifications, as in model-based testing techniques [Jacky et al. 2008]. Formal specifications can be compiled to code-based oracles that determine when a particular test passes or fails [Hierons et al. 2009]. Even after systems have been deployed, code generated from specifications can provide runtime monitoring of a system's execution and aid in the implementation of fault-recovery mechanism [Bartetzko et al. 2001; Barnett and Schulte 2003; Cheon and Leavens 2002]. Finally, in critical systems, tools based on combinations of automatic and interactive theorem proving can be used for *verification*—proving that an implementation is free of bugs and that it satisfies its formal specification in every possible execution [Barnes 1997; Cohen et al. 2009].

Formal specification can be used for many artifacts of a software development project, such as requirements, software architecture, and code. Our focus in this survey is on specification languages that can be used to record detailed design (i.e., coding) decisions about program modules. Wing [1987, 1990] and Lamport [1989] called such specifications *interface specifications*, since they document both the interface between

[3]For example, a pint means 473ml in the U.S., but 586ml in the U.K.

such modules and their behavior. However, the term interface specification tends to be confused with very weak specifications that document just the syntactic interface of various modules, such as the names and types of methods [Object Management Group 1992]. Thus, we use the term *behavioral interface specification* [Cheon and Leavens 1994] to emphasize the behavioral component of such specifications, such as pre- and postconditions. In this survey, we focus on behavioral interface specification languages. We give an overview of specification languages for other artifacts in the next section.

The main challenge in designing a behavioral interface specification language is to balance various design goals, including expressiveness, versatility, and ease of use. For instance, support for unbounded quantifiers increases the expressiveness of specifications but prevents some formulas from being checked efficiently at runtime and, thus, makes them less useful for testing. Support for higher-order logic increases expressiveness but limits the applicability of automatic program verifiers. Denoting specifications in terms of expressions of the programming language makes specifications easier to read and write for programmers but complicates verification and sometimes limits expressiveness.

During the last decade, behavioral interface specification languages have been an area of active research, driven by the development of tools that seek to improve the quality of program code. The development of these tools and their application to nontrivial programs has lead to important insights.

—The community has developed techniques for specifying properties of programs written in modern programming languages—in particular, imperative object-oriented languages.
—There is now a better understanding of the trade-offs in designing a behavioral interface specification language—in particular, the trade-off between expressiveness, annotation overhead, and support for automatic reasoning.
—There is a general agreement on a set of well-understood core concepts, as well as on current research challenges.

In this survey, we give an overview of the state of the art in designing and using behavioral interface specification languages. We focus mainly on languages and notations for expressing properties that enable or help one verify that (part of) a program satisfies certain properties. Such properties can range from the absence of certain programming faults to the satisfaction of all specified properties. We discuss many examples of such properties. To facilitate comparisons and because not all specification languages support the same features, we present examples in several specification languages, including JML [Leavens et al. 2009], SPARK [Barnes 1997], Separation logic [O'Hearn et al. 2001], and Spec# [Barnett et al. 2005]. Translations of each example into several specification languages are available online at http://www.eecs.ucf.edu/~leavens/speclang-survey/examples/index.html.

We intend this survey to be interesting for readers who want to (1) learn ways to specify or describe software properties, e.g., for use by verification or static analysis tools or (2) design a formal specification language for documenting detailed designs.

In particular, we hope that this survey will be useful to researchers participating in the Verified Software Initiative (VSI) [Hoare 2005; Hoare et al. 2007].

1.1. Classification of Specifications

Specifications can be classified according to the kinds of properties they express and the kinds of artifacts they describe. In this section, we list some of the most important kinds of properties and artifacts, and provide references for those kinds that are not covered in this article.

1.1.1. Kinds of Properties. Specification languages have focused on expressing the following three kinds of properties.

—*Functional behavior properties* describe the (data) values associated with system operations or state changes. Such properties typically describe the relationships between inputs and outputs and, thus, typically do not involve more than two states, usually the pre- and post-states of a procedure. For instance, a functional behavior specification of a sorting routine might include that the output is a permutation of the input and that the output is sorted. Functional behavior properties also include consistency criteria of data structures (such as object invariants).

—*Temporal properties* describe properties of a system's sequences of states, along with the relationship between system events and state transitions. For instance, a temporal specification of a server might include that every request is handled eventually.
Temporal properties can be expressed in temporal logic [Emerson 1990; Manna and Pnueli 1992], or event-based specification languages, such as Statemate [Harel et al. 1990], Petri nets [Peterson 1977, 1981], and process algebras [Hennessy and Milner 1985; Milne and Milner 1979; Milner 1990; Milner et al. 1992]. Specification languages that handle various forms of temporal logic include the Bandera Specification Language [Corbett et al. 2000], Promela [Holzmann 1997], and SLIC (the SLAM model checker's specification language) [Ball and Rajamani 2001].

—*Resource properties* describe constraints on how much of some resource, such as time or space, may be used by an operation or may be used between a pair of events. Timing constraints are especially important for the modeling and analysis of real-time systems. Termination is a special case of a timing property [Hehner 1989].
Resource properties can be expressed in specification languages such as timed automata [Alur et al. 1990], TPTL [Alur and Henzinger 1994], metric temporal logic [Ouaknine and Worrell 2005], HighSpec [Dong et al. 2006], CS-OZ-DC [Olderog 2008], UPPAAL [Larsen et al. 1997], Esterel [Berry 2000], Lustre [Halbwachs et al. 1992; Pilaud et al. 1987], and the duration calculus [Hansen 2008]. Timing properties for events in code can be specified in the PSpec language [Perl and Weihl 1993] and Real-time Euclid [Kligerman and Stoyenko 1992; Stoyenko 1992]. Furia et al. [2010] survey specification languages that build on temporal logic for real-time system specification.

An orthogonal way of classifying properties distinguishes between safety and liveness properties [Manna and Pnueli 1992]. A *safety* property says that nothing bad happens, for example, that the system does not crash. A *liveness* property says that something good eventually happens, for example, that a system eventually responds to a request. Functional behavior properties and resource properties are typically safety properties, whereas specifications of temporal properties often include liveness properties.

1.1.2. Kinds of Artifacts. Specifications are used to describe many artifacts in the software development process. A very broad characterization of these artifacts and their relation to known specification languages is as follows.

—*Requirements-level specifications* describe the behavior of a system's software as a whole [Jackson 1995; Heitmeyer et al. 2007; van Lamsweerde 2000].
Requirements specification languages include SCR [Heitmeyer et al. 1998], RSML and its variants [Heimdahl et al. 2003; Leveson et al. 1999], FSP [Kramer and Magee 2006; Magee and Kramer 2005], CSP [Brookes et al. 1984; Hoare 1985; Roscoe 1994], and parts of the UML [Arlow and Neustadt 2005; Rumbaugh et al. 1999], such as Statecharts [Harel 1987].

—*Analysis-level specifications* describe domain models and express concepts in a domain.
Analysis specification languages that describe functional behavior include Alloy [Jackson 2006], Z [Hayes 1993; Spivey 1989], TLA [Lamport 1994], ASML [Börger and Stärk 2003; Gurevich 1991], and algebraic equational specification languages [Bidoit et al. 1991; Goguen et al. 1978; Wirsing 1990], such as LSL [Guttag and Horning 1986] and CASL [CoFI (The Common Framework Initiative) 2004; Mossakowski et al. 2008]. Some refinement-oriented languages, such as B [Abrial 1996], have the capability to state specifications at this level and also lower levels.
—*Architectural specifications* describe the intended connections (control and data flows) between components of a program [Aldrich et al. 2002; Garlan et al. 2000].
—*Code interface specifications* describe contracts between the implementation of a program part and its clients. These specifications can be expressed both as types and as more general assertions. Type systems are convenient ways to state invariant properties that are true in all states. Types can concisely state a finite set of invariant properties, such as non-nullness and numerical range restrictions [Nielson 1996] and can also encode general predicates with the aid of dependent type constructors [Backhouse et al. 1989; Constable et al. 1986; Schmidt 1994; Martin-Löf 1985].
Languages for functional properties rely heavily on one or two-state assertions, including pre- and postconditions, as well as invariants. This style of specification language is typified by Eiffel [Eiffel 2005] and includes languages such as Gypsy [Ambler et al. 1977], Anna [Luckham and von Henke 1985; Luckham 1990], SPARK [Barnes 1997; Chapman 2000], VDM [Andrews 1996; Fitzgerald and Larsen 1998; Fitzgerald 2008; Jones 1990], VDM++ [Fitzgerald et al. 2005; Mitra 1994], Larch interface specification languages [Guttag et al. 1993; Guttag et al. 1985; Wing 1987], the RESOLVE family [Ogden et al. 1994; Edwards et al. 1994], Spec# [Barnett et al. 2005; Barnett et al. 2006], and JML [Burdy et al. 2005; Leavens et al. 2006; Leavens 2006]. The Object Constraint Language of the UML [Warmer and Kleppe 1999; OMG 2006] also fits in this style.

In this survey, we focus on code interface specification languages for functional behavior properties. However, some of these properties are also relevant for proving temporal or resource properties. For instance, proving termination of a list traversal might depend on an invariant that the list is acyclic.

1.2. Specification Languages and Verification Technology

Specification languages are often specialized to support some particular automated verification technique. An example is Promela's support for the SPIN model checker [Holzmann 1991]. However, there are interesting synergies between several verification techniques, such as static and dynamic checking [Ernst 2003; Flanagan 2006], and some specification languages, such as JML [Burdy et al. 2005], have been designed to support multiple tools.

Verification technology is also closely tied to semantics. A *verification logic* is a formal reasoning system that allows proofs that establish that code satisfies a specification [Apt 1981; Apt and 0lderog 1991; Bjørner and Henson 2008; Cousot 1990; Emerson 1990; Francez 1992; Hoare 1969; Kozen and Tiuryn 1990; Manna and Pnueli 1992]. While it is, in principle, possible to directly use program semantics to specify and verify programs, it is often more convenient to encapsulate reusable proof principles for a given programming language in a verification logic. Then one uses the verification logic to prove correctness. That such proofs are sound (or valid) is proved using the programming language's semantics [Apt 1981; Apt and Olderog 1997; Loeckx and Sieber 1987; Winskel 1993]. On the other hand, model checking [Clarke et al. 1986;

Holzmann 1997] uses state space exploration techniques to verify programs, which is directly based on the semantics of finite state machines.

In this survey, we make special efforts to highlight the interplay between specification language design and automated verification techniques. Designing a specification language to fit a particular verification technique can be an excellent way to make that verification technique widely available and easily usable. Conversely, one important aim of verification technology is to support clear and precise communication with human readers; thus, new specification language features provide interesting challenges to those interested in verification technology.

1.3. Outline

In the remainder of this survey, we give an overview of the main specification techniques for the functional behavior of sequential programs. (We largely omit consideration of concurrent and parallel programs due to lack of space.) In the next section, we discuss specification of inputoutput behavior of subprograms, including contracts for procedures. Section 3 discusses constructs for specifications of modules, including packages and abstract data types. Section 4 describes the specification of data structure shapes and how to specify procedures that manipulate data structures. Section 5 describes how specification languages treat subtyping and dynamic dispatch as found in object-oriented programming. Finally, Section 6 offers some conclusions.

2. INPUT/OUTPUT BEHAVIOR

The most basic notions of behavioral specifications capture a program's (or subprogram's) input/output behavior. For example, an input/output specification of a sorting algorithm might capture the fact that the algorithm takes as input an arbitrary array of integers and returns a permutation of the input array with the elements arranged according to some total order. Input/output specifications may also constrain the intermediate states that a program passes through in the process of transforming inputs into outputs. One common form of constraint specification is an *assertion*—a predicate embedded in the code that must hold on the current program state when execution reaches the point at which the assertion is written. For example, in a sorting algorithm, an assertion might be used to state that the array being sorted must be nonempty before the sorting process begins. Another common form of constraint is an *invariant*—a predicate that must hold each time program execution reaches one of a predefined set of program points. For example, in an insertion-sort program, one might have an invariant stating that, whenever program execution reaches a new iteration of the outer loop, all elements to the left of the loop index are in sorted order.

This section gives an overview of basic forms of code-level input/output specifications. Although many of the forms of specification that we discuss are relevant in a variety of programming language paradigms, we focus our initial discussion of specification concepts in simple imperative languages. A program $\mathcal{P}$ in such a language consists of commands, such as variable assignment, conditionals, and loops, organized into subprograms, such as procedures and modules. We will discuss more advanced language features in later sections.

The SPARK language [Barnes 1997] is a nice example of a simple imperative language with code-level specifications and associated checking tools. The SPARK programming language is a subset of Ada designed for programming high-assurance applications, and it omits features, such as heap-based data and exceptions, that often cause difficulty when reasoning about programs. The SPARK specification language includes a variety of annotations and first-order logic expressions that are embedded

in Ada comments within the SPARK code. We use SPARK in this section to illustrate many of the concepts associated with input/output specifications.

2.1. Inline Assertions

2.1.1. Overview. Consider a situation in which client code uses a library function to generate a random number. Let us assume that the client developer reads the informal documentation associated with the library and concludes that the generator will always return nonnegative values. The developer can use an assertion to specify this assumption/expectation about the behavior of the library function at the point where the random function is used.

```
R := Lib.Random();
--# assert R >= 0;
```

Assertions are typically written as side-effect-free boolean expressions in the programming language in which they are embedded. In some cases, assertions are embedded in program comments (as with the preceding SPARK assertion, which is embedded in an Ada inline comment delimited by `--`) and are recognized and processed by verification tools that understand the special comment syntax (`#` appended to the Ada comment delimiter `--#` indicates a SPARK specification). In other cases (such as with Java assertions), assertions are written as executable code that may be executed during testing and removed by a preprocessor once testing is completed.

An assertion documents an expectation. By expressing assumptions and important aspects of the intended functionality of the code, assertions can thus aid in code maintenance. Typically, assertions are also viewed as executable specifications. When an assertion is executed, its evaluation has no effect if the assertion holds for the current state. If the assertion does not hold, program execution is halted and the system generates an error message.

2.1.2. Tool Foundations. Because the assertion concept is easy to learn and easy to use, it one of the most familiar, widely used forms of program specification (for example, McConnell's *Code Complete* [McConnell 1993]—one of the most widely read practical guides to writing commercial software—emphasizes using assertions in defensive programming). The evolution of assertions and associated checking tools has a rich history. Floyd made very early use of the assertion concept as a program specification mechanism [Floyd 1967]. Hoare's seminal paper on axiomatic semantics [Hoare 1969] also makes fundamental use of assertions. Two pioneering languages that were designed for verifying programs, Gypsy [Ambler et al. 1977] and Euclid [Lampson et al. 1981], used `assert` statements (and other specifications) as built-in language features. Due to their increasing use in current programming practice, modern languages like Eiffel, Java, and Spec# also include assertions as built-in language features. However, inline assertions can be included in almost any language by defining an `assert` construct as a procedure or macro. For more details on the history of assertions and assertion checking, we refer the reader to the work of Jones [2003] who surveys the early use of assertions in formal reasoning about programs to and Clarke and Rosenblum [2006] who provide a detailed historical perspective on the development and use of runtime assertion checking.

Assertions may be checked statically using automated deduction techniques or at runtime, as the program is executed. For example, ESC/Java [Flanagan et al. 2002]—one of the first tools to provide checking of specifications for realistic Java programs—attempts to prove that each inline assertion will hold in all executions using decision procedures and weakest precondition calculations as the underlying reasoning technology. Similar technology is used for checking assertions in the Spec# verification

framework for C# [Barnett et al. 2005]. Symbolic execution, which combines decision procedures with strongest postcondition calculations, is another popular approach for assertion checking used in tools such as Java PathFinder [Khurshid et al. 2003] and Kiasan [Deng et al. 2006] for Java, Cute [Sen et al. 2005] for C, and XRT [Grieskamp et al. 2005] for Spec#. These tools aim for completely automated checking of assertions and, thus, provide checking on only bounded portions of a program's statespace or report false alarms—situations were assertions that are actually valid are reported as failing due to approximations made during checking. Other frameworks, such as SPARK and the C verifier VCC [Cohen et al. 2009], aim to provide complete verification by generating verification conditions from assertions, discharging as many verification conditions as possible with no user intervention using decision procedures, and then requiring users to employ interactive theorem provers to discharge the remaining verification conditions.

In static checking of assertions, it is useful to distinguish between assertions that one expects a program checker to verify and assertions that cannot be verified in the given context but provide logical facts that are needed for the rest of a static verification. Following the refinement calculus [Back 1980; Morgan 1990], some specification languages offer a way to make this distinction, using `assert` statements for the former and `assume` statements for the latter. The distinction between `assert` and `assume` only manifests itself in the behavior of the program verifier. When encountering `assert B`, the verifier checks that the boolean expression `B` holds. If it does not, the verifier reports an error and execution along that verification path is halted. If `B` holds, then the verifier continues exploring the current execution with the set of logical facts known by the verifier enriched to include `B`. When encountering `assume B`, the verifier continues, after adding `B` to the set of known logical facts.

For example, the following `assert` statements (written in the syntax of VCC [Cohen et al. 2009]) express the intention for the code to compute the minimum of two numbers and instruct the program verifier to prove that the code (which uses arithmetic tricks to avoid branches) is correct.

```
m = a - ((a - b) & -(a > b));
_(assert m <= a && m <= b)
_(assert m == a || m == b)
```

In this case, if the verifier cannot prove one of the assertion conditions using its current fact set, an error is reported. In contrast, consider this example using the library function `Lib.Random`, and assume that the source code is not available for the function. Without a formal specification of the function, it is not possible to prove the assertion `R >= 0` that occurs after the call, because there no information about the possible return values of the function. In a specification language that supports the verify-versus-assume distinction, one would instead use an `assume` statement (here, shown in Spec# [Barnett et al. 2005]).

```
R = Lib.Random();
assume R >= 0;
```

The verifier does not try to prove `R >=0` but, instead, simply continues to explore the current execution path with the fact `R >=0` added to its fact set.

Inline assertions can also be checked at runtime by compiling a representation of each assertion into executable code. The resulting assertion-instrumented code can be effective in finding unexpected behavior during the development and testing of a program [McConnell 1993]. The compiled representation of assertions does introduce computational overhead into runtime execution, and when static checking of assertions is combined with runtime checking, overhead can be reduced by omitting code

generation for assertions that have been proven statically to always hold (code generation for `assume` statements checks would be maintained if strict conformance to their semantics is desired, since they are not proven statically). In addition, when the quality of a program is high enough to deploy it, and soundness of `assume/asserts` need not be strictly enforced, then code generation for assertions can be disabled to trade the additional checking time for improved performance on the program's main tasks.

2.1.3. Partial Expressions. Syntactically, it is possible to write an assertion whose evaluation is not well defined. For example, the assertion `x/y = 5` is not defined in states where `y = 0`. There are five general approaches to dealing with this problem.

One approach is to give expressions a nonstandard interpretation that treats all functions and operators as total when they appear in an assertion context. For example, `x/0` would then denote some unspecified function of `x`, maintaining the property that an expression is always a function of its subexpressions. Because the particular value returned by `x` is unspecified, it is possible to prove the assertion only if it is immaterial what the value of `x` is. This approach has been taken, for example, in the Larch Shared Language and in ESC/Java. It was also adopted in the JML runtime assertion checker, which went to great lengths to convert evaluation failures for boolean-valued expressions into *false* or *true* in an attempt to make undefinedness produce assertion violations [Cheon and Leavens 2005]. However, while this approach has supporters [Gries and Schneider 1995], it has been criticized, for example, because it has some possibly undesirable consequences in typed logics [Jones 1995], and because it requires expressions to be interpreted differently when they appear in an assertion context as opposed to a program context [Chalin 2005].

Another approach is to treat any undefined subformula of an assertion as the value *false* [Parnas 1993]. For example, `x/y = 5` is then treated as *false* whenever `y = 0`. A problem with this approach is that `x/y = 5` might then yield the same result as `x/y` $\neq$ `5`; indeed, `x/y = x/y` is not always *true*.

The two preceding approaches admit undefined expressions. That is, they do not assign any blame for having written an assertion that is not always defined. The next two approaches will cause an error to be reported for undefined evaluations.

The third approach to dealing with undefined expressions in assertions is to generate an error at the time the assertion needs to be checked. This approach is taken, for example, in Eiffel. Although simple, a consequence of the approach is that undefinedness errors in assertions may go undetected for a long time. In addition, the undefinedness check is repeated many times during runtime checking.

The fourth approach is to demand that assertions be well defined in all possible contexts. This can be enforced in a static program verifier by generating a well-definedness check for every assertion given in the program. A consequence of this approach is that assertions must be written to be *self guarding* or *protective* [Leavens and Wing 1997; Rudich et al. 2008]. For example, an assertion that mentions `x/y = 5` must also explicitly check for `y` $\neq$ `0`, as in `y` $\neq$ `0` $\Rightarrow$ `x/y = 5`. This approach is compatible with programming languages that have short-circuit operators [Chalin 2007]. For example, in Java, using the `||` operator one can write `y == 0 || x/y == 5`.

The fifth approach is to abandon ordinary first-order logic and, instead, to consider a logic that deals with partial expressions directly. One such logic is the Logic of Partial Functions [Barringer et al. 1984]. Advantages of this logic are argued by Jones [Jones 2006] in a paper that also gives a broader history and evaluation of different approaches found in the rich literature on the subject of partial expressions. Woodcock et al. [2009] have combined the approaches to partial expressions in various logics into a unifying theory, which lets them be compared and (under certain restrictions) be used interchangeably.

```
function Find_Index_Pos (X : in Integer) return Index_Range
--# global in S;
--# pre for some M in Index_Range => (S(M) = X);
--# return Z => (S(Z) = X) and
--#    (for all M in Index_Range
--#       range Index_Range'First .. Z-1
--#     => (S(M) /= X));
is
  Result_Pos : Index_Range;
begin
   for I in Index_Range loop
      if S(I) = X then
         Result_Pos := I;
         exit;
      end if;
   end loop;
   return Result_Pos;
end Find_Index_Pos;
```

Fig. 1. SPARK illustration of structured assertions.

2.2. The Pre/Post- Technique

2.2.1. Overview. While assertions can be placed at arbitrary points in the code, they can also be used in a structured manner to enable more systematic reasoning about program behavior. Pre/postconditions are one example of structured use of assertions. A precondition is an assertion that must hold whenever the procedure is called (after parameter passing). A postcondition is an assertion that must hold immediately after the procedure completes its execution.

The SPARK procedure[4] `Find_Index_Pos` of Figure 1 illustrates the use of pre/postconditions to express that the procedure searches a global array `S` for the value of input parameter `X`. First, note that the assertions representing pre/postconditions are not stated in the procedure body using the `assert` keyword, as in the previous example. Instead, to highlight the distinguished role of the pre/postcondition assertions, they are written using the keywords `pre` and `return` and placed in a special collection of annotations in the header of the procedure.

The precondition states that the value of `X` must be in the array when the procedure is invoked. The use of existential quantification in the precondition (i.e., there is some position `M` in the array that holds the value `X`) illustrates that the language of assertion expressions may, in some cases, be richer than the language of program expressions.

In the postcondition, the construct `return Z` names the return value `Z` and imposes two constraints: (1) the value at index position `Z` is equal to `X`, and (2) `Z` is the first position in the array to have a value of `X`.

Together, the pre- and postconditions of a procedure can be viewed as summarizing the procedure's input/output behavior in the sense that the associated assertions describe properties of states flowing into and out of the procedure. Summaries can vary in their precision. For example, dropping the constraint (2) in the postcondition just listed still yields a valid summary of the associated implementation, but it is a less precise summary, because it does not capture the fact that the index value returned corresponds to the first occurrence of `X`.

It is also fruitful to view a pre/postcondition pair as defining a *contract* κ between a procedures P and its clients (other procedure that call P). From the point of view of a client of P, the client must abide by the contract by calling P in a state that satisfies

[4]Our discussion ignores the SPARK/Ada distinction between functions and procedures.

κ's precondition. When doing so, it can rely on P to satisfy the contract by completing in a state that satisfies κ's postcondition. From the point of view of the implementation of P, the implementation can assume that it will always be called with parameters and an associated global variable state that satisfies κ's precondition. Working under this assumption, P must fulfill its contract by ensuring that κ's postcondition will always be satisfied. The term "Design by Contract" was coined by Meyer [1992] to describe a program methodology that emphasizes the contract metaphor by encouraging early definition of program module contracts with coding following at a later stage, guided by previously constructed contracts.

2.2.2. Semantic Foundations. The use of pre/postconditions in program specification can be traced back to Floyd/Hoare logic [Floyd 1967; Hoare 1969] which characterizes the behavior of a program statement C using triples of the form

$$\{P\}\, C\, \{Q\},$$

where both P (the precondition) and Q (the postcondition) are boolean formula over variable values.

A *state* σ for a program is a mapping from the program's variables to values. We call a boolean formula over variable values a *state predicate* and say that a state σ *satisfies* a state predicate P when the values of variables, as given by σ, satisfy the constraints given by P (i.e., when $P(\sigma) = true$). There are two common ways to define the semantics of Hoare triples, which differ only in their treatment of non-termination.

Total correctness. A Hoare triple is *valid* if and only if every execution of the program C starting in a state satisfying precondition P will terminate in a state satisfying postcondition Q.

Partial correctness. A Hoare triple is *valid* if and only if every execution of the program C starting in a state satisfying precondition P will run forever or terminate in a state satisfying postcondition Q.

Differences in precision of summaries, such as those discussed for the example of Figure 1, can also be captured within this logical view. A formula Q is weaker than P if $P \Rightarrow Q$ (P entails Q). When $P \Rightarrow Q$, we say that Q *abstracts* P (equivalently, P *refines* Q). Intuitively, this means that Q is less restrictive and more approximate than P, and P represents a more precise summary of states. State predicates can be viewed as abstractions that summarize state information, and they can be arranged in a natural approximation lattice based on the entailment relation as an ordering.

This preorder of approximation on state predicates can be used to define a preorder on pre/postconditions pairs (procedure contracts), as illustrated in the following diagram.

$$\begin{array}{lcccl} \textit{less precise} & & \{P\} \cdot \{Q\} & & \\ & \textit{refined by} \downarrow & \Downarrow \quad\quad \Uparrow & & \textit{abstracted by} \uparrow \\ \textit{more precise} & & \{P'\} \cdot \{Q'\} & & \end{array}$$

Let κ represent the contract $\{P\} \cdot \{Q\}$ and κ' represent the contract $\{P'\} \cdot \{Q'\}$. κ' is said to be a *refinement* of κ (alternatively, κ is an *abstraction* of κ') if $P \Rightarrow P'$ and $Q' \Rightarrow Q$ [Back 1980]. From the point of view of clients, if κ' is a refinement of κ and if C and C' satisfy κ and κ', respectively, then C' can be used in any context where C is used. This is because C' imposes stronger conditions on its output while being more permissive on inputs then C [Chen and Cheng 2000; Naumann 2001; Olderog 1983]. Any context that can supply inputs satisfying C's precondition P can also supply inputs satisfying C''s precondition P', since $P \Rightarrow P'$. Similarly, any context that can accept outputs satisfying C's postcondition Q can also accept outputs satisfying C''s postcondition

```
package BoundPackage
is
   Array_Size : constant := 10;
   type Index is range 1 .. Array_Size;
   type IntArray is array(Index) of Integer;

   procedure UpperBound(A : in IntArray; Result : out Integer);
   --# derives Result from A;
   --# pre (for all I in Index => (A(I) >= 0));
   --# post (for all I in Index => (A(I) <= Result));

   procedure LeastUpperBound(A : in IntArray; Result : out Integer);
   --# derives Result from A;
   --# pre true;
   --# post (for all I in Index => (A(I) <= Result)) and
   --#      (for some J in Index => (A(J) = Result));
end BoundPackage;
```

Fig. 2. A SPARK example illustrating contract refinement.

Q, since $Q' \Rightarrow Q$. This substitutability is also used to reason about object-oriented programs with dynamic method binding, as we explain in Section 5.

Figure 2 shows SPARK package (a module) specification with two procedures. The contract for `LeastUpperBound` refines the contract for `UpperBound` because (1) it has a weaker precondition (it doesn't impose any constraints on the prestate whereas `UpperBound` requires all array elements to be `>= 0`), and (2) it has a stronger postcondition (it not only requires `Result` to be an upper bound of the array elements, it also requires that `Result` is equal to one of the array elements). Therefore, it is sound to replace any call to `UpperBound` by a call to `LeastUpperBound` instead. Every caller that lives up to the precondition of `UpperBound` also satisfies the weaker precondition assumed by `LeastUpperBound`, and `LeastUpperBound` guarantees a postcondition that implies the postcondition assumed by a caller of `UpperBound`.

2.2.3. Tool Foundations. Early tools using Floyd/Hoare logic required a high degree of manual intervention to construct appropriate pre/postconditions. However, modern tools achieve significant amounts of automation using techniques such as *weakest precondition calculation*. A *weakest precondition operator* $wp(C, Q)$ takes a command C and postcondition Q and automatically constructs a precondition P that makes $\{P\}\,C\,\{Q\}$ valid. More precisely, P is constructed to be the weakest formula that can establish Q as a postcondition for C [Dijkstra 1976]. Recalling the preceding discussion of weakest, the precondition returned by $wp(C, Q)$ is the most general (or best) one in the sense that it imposes the fewest restrictions on inputs to C that can guarantee Q to hold. A *wp-calculus* contains rules for computing *wp* for each command of the programming language.

2.2.4. Relational Postconditions and Two-state Predicates. It is common that a postcondition describes not only what holds in the post-state of a procedure but also how the post-state relates to the pre-state of the procedure. A simple example is an increment procedure whose postcondition says that a variable `x` is 1 more in the post-state than in the pre-state. There are two general approaches to dealing with such relational postconditions.

One approach is to introduce a new notation for the pre-values (or post-values, or both) used in postconditions. Figure 3 illustrates this approach with an increment procedure written in SPARK, where `x~` denotes the pre-value of the global integer variable `x`. The notation `old x` is used for pre-values in Eiffel (which influenced JML

```
procedure Inc()
--# global in out x;
--# post x = x~ + 1;
is
begin
  x := x + 1;
end Inc;
```

Fig. 3. SPARK example of a relational postcondition.

and Spec# to do the same), and x_0 is used in Morgan's specification statements [Morgan 1990]. Transition systems (e.g., TLA [Lamport 1994]) and relational program logics (e.g., [Hehner 1993; Hoare and He 1998]) often use the notation `x'` to denote the post-value of variable `x` and `x` to denote the pre-value.

The other approach to specifying relational postconditions is to make use of *logical variables*—variables that are used only in specifications and cannot be assigned by programs. For example, in some specification language with logical variables in which `pre` and `post` are keywords for pre/postconditions, the specification

$$\texttt{pre}\ \texttt{x} = \texttt{X}; \qquad \texttt{post}\ \texttt{x} = \texttt{X} + 1;$$

says that if program variable `x` has initial value `X` (where `X` is a universally quantified logical variable whose scope is this procedure specification), then it will have the final value `X + 1`. Logical variables are sometimes natural to include in specifications, especially if they denote some value for which there is no unique or convenient pre-state expression; for example, `X` might be introduced in a precondition like `x < X && X < y` or `hash(X) = y`. They are also easily handled when verifying the procedure implementation, where there is no difference between logical variables and other variables that happen not to be assigned.

The meaning of a logical variable at a call site is more delicate. First, there is a proof obligation, typically imposed at call sites, that there must exist a value for each logical variable mentioned in the precondition. Second, the specification says that the postcondition holds for all values of the logical variables allowed by the precondition. Logical variables have been used in logics like Hoare logic [Hoare 1969] and separation logic [Reynolds 2002] and in specification notations like Z [Spivey 1989].

2.2.5. Framing. An important piece of a procedure specification is the piece that indicates which parts of the program state the procedure is allowed to modify. This piece is called the *frame* of the procedure, and it is what, for example, allows callers to determine which parts of the caller state are not modified by the call [Borgida et al. 1995]. There are three general approaches to specifying framing.

One approach is to explicitly write in the postcondition what is not modified, implicitly saying that all other variables may change. This can be done with logical variables or `old` expressions (see Section 2.2.4) by explicitly stating for each unchanged variable that the value of the variable in the post-state is equal to the pre-state value of the variable. This approach is problematic when some variables are not in scope (and, hence, cannot be explicitly mentioned) in the specification. This has led to some intricate theorems of Hoare logic-style *frame rules* [Hoare 1971; Reynolds 1981; O'Hearn et al. 2001; Banerjee et al. 2008] and issues like fully abstract semantics [Meyer and Sieber 1988]. We will say more about such information-hiding concerns in Section 3.

Another approach to framing is to include a specification construct that indicates what may be modified, implicitly saying that other variables are not modified. For example, in a program with two variables `x` and `y`, using the `modifies` construct of Liskov and Guttag [1986] one can write `modifies x` to specify that `x` may be modified and that

y is unchanged. Forms of `modifies` clauses have been used in many specification languages, including the Larch family, JML (which uses the keyword `assignable`), Spec#, and Z (which uses the symbol Ξ).

An extension of this approach is the variation used in SPARK. Each SPARK procedure may reference or update the state associated with its parameters, as well as that of global variables. To capture framing, Figure 1 illustrate that each SPARK procedure contract must explicitly list the global variables accessed (both reads and writes) during procedure execution in a `globals` construct. Moreover, for each parameter and global variable, *mode annotations* `in`, `out`, and `in out` must be used to indicate if each parameter/global in read only, written only, or both read and written.[5] For example, the global variable `x` of the `Inc` procedure in Figure 3 is read-write, whereas `S` of procedure in Figure 1 is read only. Together, the mode annotations on parameters and globals can be viewed as giving a complete specification of the inputs and outputs of a procedure.

SPARK framing specifications are more precise than those based on a `modifies` clause, because SPARK's mode annotations enable one to describe more precisely how a variable is accessed. Specifying `in`/`out` modes is more relevant in languages like Ada where parameters can be passed by reference.

The third approach to framing is to let the precondition, in concert with the program semantics, limit what a procedure can modify. This approach is taken in separation logic and Implicit Dynamic Frames [Smans et al. 2009], where (reading and) writing to memory requires knowing that the memory contains that location, which ultimately comes down to the procedure precondition having to specify this. By analyzing what memory locations the precondition depends on, a caller can figure out an upper bound on the effect of the call. For example, the separation logic precondition

$$\mathtt{x} \mapsto \mathtt{X} \quad \wedge \quad 0 \leq \mathtt{X}$$

says that `x` points to a memory location that holds a nonnegative number. The precondition implicitly tells callers that other memory locations are unchanged. We discuss framing and separation logic in more detail in Section 4.

2.3. Iteration and Recursion

Reasoning about loops and recursion is often challenging, because one must write specifications that capture the effect of repeated computations where the number of repetitions is not known in advance. In these cases, reasoning usually proceeds according to some induction principle, and thus one aims to specify a property that is preserved by each repetition of the computation. Another challenge for loops and recursion is to specify termination arguments.

2.3.1. Loop Invariants. When reasoning about loops, properties that are preserved by each repetition of the computation are called *loop invariants*. A loop invariant is a state predicate that always holds (i.e., is invariant) each time execution reaches the top of the loop. The role of loop invariants in specifying functional properties of loops can be seen in the Hoare logic rule for **while** loops [Hoare 1969].

$$\frac{\{P \wedge b\}\, C\, \{P\}}{\{P\}\, \mathbf{while}\ b\ \mathbf{do}\ C\, \{P \wedge \neg b\}}$$

Consider a situation in which we want to prove that a given loop concludes with a set of variables satisfying Q. We must then find a sufficiently strong predicate P that can be shown to be an invariant of the loop. Proving P to be a loop invariant is done by showing two properties. First, one shows that P holds when the program reaches

[5]VDM contains similar annotations [Jones 1990].

the loop. This proves that P holds the first time the top of the loop is reached. This corresponds to the base case of an inductive argument. Second, one shows that P is maintained by the loop body. This checks that P holds at the end of the loop body, which implies it holds when loop execution branches back to the top of the loop. It corresponds to an induction step.

If, at the top of the loop, the loop guard holds, then a new iteration is started. Hence, when proving the loop body to maintain P, the loop guard b can be assumed, as shown in the triple above the line in the preceding Hoare logic rule above. If the loop guard does not hold, the loop terminates, and thus, the loop invariant is strong enough to prove the desired Q if $P \wedge \neg b$ implies Q.

Due to basic undecidability results, an algorithm for calculating weakest preconditions for while loops that can be discharged by automated theorem provers is infeasible [Cousot 1990]. Useful automated approaches that find or over-approximate invariants for loops exist for restricted classes of data values (using techniques such as abstract interpretation [Cousot and Cousot 1977]), but many program verification tools require loop invariants to be explicitly stated or even take the unsophisticated approach of sacrificing soundness by verifying only a bounded number of loop iterations.

Flexible Loop Exits. We have shown the treatment of loops for common while loops, where the one and only exit point is the loop guard that is evaluated at the top of the loop. Standard approaches extend to loops with more flexible exit points, like repeat-until loops and loops with `exit` (as in Ada, or `break` as in Java) statements. The program logic for such loops still applies the loop invariant at the top of the loop, but instead of using the condition $P \wedge \neg b$ on loop exit, it is as if the exiting loop iteration starts in a state satisfying P and then, after executing the intervening statements, exits at the appropriate control point. More formally, to treat such loops, it can be convenient to use an underlying semantics that allows jumps [Cristian 1984; Manasse and Nelson 1984; Leino et al. 1999; Huisman and Jacobs 2000].

The example in Figure 4 shows a JML specification of a loop containing a `break` statement. Method `ValuePresent` returns *true* when the integer `X` is contained in the array `S`, which is stored in a field of the enclosing class. Since no precondition is explicitly stated for this method, the precondition defaults to *true*. The postcondition requires that the return value is *true* if and only if there is an index `M` for array `S` such that the value in `S` at position `M` is equal to the input parameter `X`. Note that the second loop invariant does not hold when the loop is exited via the `break` statement. This is sound because the loop invariants are not assumed to hold when leaving the loop via this exit point.

Some specification languages, like SPARK, treat any inline assertions found in the loop body as a loop invariant, and this results in a proof obligation structure that differs from the preceding Hoare rule for `while` loops.. For example, consider the SPARK version of `Value_Present` in Figure 5 with an assertion specifying an invariant at the end of the loop. The SPARK proof obligations for `Value_Present` would be to show that (a) under the assumption that the precondition holds, if the loop is entered (which it will be, because all ranges in SPARK must be nonempty), then the assertion holds when encountered; (b) assuming the assertion holds, entering the loop again will again lead to the assertion being satisfied and (c) assuming the assertion holds, exiting the loop will lead to the postcondition being satisfied. Thus, the specification goal is simply to cut the control paths through the loop with at least one assertion that will allow these proof obligations to be discharged. From another point of view, the semantics for the simple `Value_Present` loop is equivalent to first back-propagating the assertions

```
/*@ ensures \result ==
  @         (\exists int M; 0 <= M && M < S.length; S[M] == X);
  @*/
boolean ValuePresent(int X) {
  boolean Result = false;
  //@ maintaining 0 <= I && I <= S.length;
  //@ maintaining !Result;
  //@ maintaining (\forall int M; 0 <= M && M < I; S[M] != X);
  for(int I = 0; I < S.length; I++)
  {
    if(S[I] == X) {
      Result = true;
      break;
    }
  }
  return Result;
}
```

Fig. 4. A JML example of flexible loop exits. In JML, the special comment symbols `//@` and `/*@ ... @*/` indicate assertions. The loop invariants follow the keyword `maintaining`. Backslashes are used to prefix expression keywords in JML to avoid interference with program identifiers. Assertions occur before the program element they specify; for instance, the postcondition occurs before the method, and the loop invariant occure before the loop. In postconditions, `\result` stands for the value returned by the method.

```
function Value_Present (X : Integer) return Boolean
--# global in S;
--# return for some M in Index_Range => (S(M) = X);
is
   Result : Boolean;
begin
   Result := False;
   for I in Index_Range loop
      if S(I) = X then
         Result := True;
         exit;
      end if;
      --# assert I in Index_Range and
      --#    not Result and
      --#    (for all M in Index_Range range Index_Range'First .. I
      --#        => (S(M) /= X));
   end loop;
   return Result;
end Value_Present;
```

Fig. 5. The SPARK version of the example in Figure 4.

(using weakest preconditions) to the top of the loop and then applying the conventional Hoare `while` rule.

Loop Framing. An important issue in reasoning about loops that is often neglected in cursory treatments is *loop framing*, that is, specifying what the loop modifies. For example, when reasoning about the loop in function `Value_Present` (Figure 5), it is important to know that `S` is not changed, whereas the loop body does have an effect on `I` and `Result`.

A simple approach that goes a long way is for the semantic reasoning engine to compute the *syntactic loop targets* by syntactically scanning the body of the loop and noting which variables appear in l-value positions. All variables that are not syntactic

loop targets are then known not to be modified by the loop, and so the reasoning rule can take that into consideration. For example, if the syntactic loop targets of C does not include a variable x, then the preceding Hoare logic rule can be altered to

$$\frac{\{P \wedge b\}\, C\, \{P\} \qquad x \text{ is not among the syntactic loop targets of } C}{\{P \wedge x = X\}\, \textbf{while}\ b\ \textbf{do}\ C\, \{P \wedge \neg b \wedge x = X\}}$$

where X is a logical constant.

Like procedure framing (see Sections 2.2.5 and 4), loop framing becomes more involved when it is necessary to know that the global memory or heap is modified only at certain locations. A solution that works well in practice is to enforce the enclosing procedure's frame condition also as an invariant of the loop. This was done in ESC/Modula-3, which called it Loop Modification Inference [Detlefs et al. 1998], and is used in the verifiers for, for example, Spec# and Dafny [Leino 2010]. Another approach, which can be used by itself or to complement loop modification inference, is to allow explicit `modifies` clauses on loops. This latter approach has the advantage of the modifies clause being more restrictive than the procedure's. Such a loop `modifies` clause could also be more permissive to allow the loop to make modifications that are later undone (before the procedure returns). This approach was taken in the Krakatoa tool [Marché and Paulin-Mohring 2005]; JML has adopted this by use of its `refining` statement, which can give arbitrary specifications (including frames) to statements [Shaner et al. 2007; Leavens et al. 2009].

Finally, another aid in loop framing is the ability to refer to the value of variables before the loop. For example, if the specification language makes it possible to refer to the value of a variable in a loop at the time the loop was reached e.g., like `preloop(x)` then an invariant such as `x = preloop(x)` says that the loop does not change `x`. A slightly more flexible solution is to allow a program to label statements and to refer to the value of a variable from the previous encounter with that label (e.g., the JML notation `\old(x,L)`, which refers to the value of variable `x` at label `L` [Heym 1995; Leavens et al. 2009]).

Ghost variables can also be used to specify loop framing. A ghost variable is a specification-only variable that is updated by explicit (specification-only) assignment statements.[6] For example, by introducing a ghost variable `oldx` and using the statement `oldx := x` just before the loop, the invariant `x = oldx` specifies that the loop does not change `x`.

Effectively, `preloop`, `\old`, and ghost variables make loop invariants into multi-state predicates, which of course also makes it possible to write more general invariants, like `x <= preloop(x) + c`.

Alternatives to Reasoning with Loop Invariants. Although most common, not all program logics make direct use of a loop invariant. For example, in Dynamic Logic [Kozen and Tiuryn 1990], which is used by the KeY system [Beckert et al. 2007], the user needs to perform an inductive proof on the execution of the given loops. This avoids the meta argument that justifies using a loop invariant in something like the Hoare logic rule, instead, giving the user the flexibility and burden of setting up the induction.

[6]There is some inconsistency in the literature about the use of the terms ghost variables, auxiliary variables, and logical variables. Susan Owicki's thesis [Owicki 1975] uses "auxiliary variables" for what we have called "ghost variables." Reynolds originally coined the term "Ghost variable" for variables that only appear in the assertions and are never mentioned in the program [Reynolds 1981]. In this survey we call this kind of variable a "logical variable." Reynold's used logical variables for variables of boolean type following Pascal. There is also literature that uses "auxiliary variable" to mean a variable not mentioned in the program [Apt 1981].

As we have shown, loop invariants specify the state at the top of the loop, which can be viewed as describing the work or state that has been achieved so far by the loop. There is some evidence that this does not always lead to the most straightforward conditions; it can be more straightforward, instead, to focus on the work that has yet to be done [Hehner 2005]. For example, such a loop specification has a postcondition `z' = z + x*y`, which expresses that what must be achieved is a new state in which z's value (written `z'`) is increased by the current value of `x*y`.

2.3.2. Termination. Repeated computations give rise to potential non-termination. Loop invariants and pre/post specification allow one to prove the partial correctness of loops and procedures. To enable proofs of termination and, thus, total correctness, specifications need to express termination arguments for loops and recursive procedures.

Loop Termination. A common approach to proving that a loop does not iterate forever is to give a function, called a *bound function* or *variant function*, from the loop-head state to a value in a well-founded order and to show that successive values in any execution strictly decrease. Another approach is for the semantics to model the number of completed loop iterations (or some other measure of the passing of time) in a ghost variable and to prove the loop invariant that this variable is bounded [Hehner 1998].

For example, to specify a variant function for the loop in method `ValuePresent` (Figure 4), one might add the following clause.

```
//@ decreasing S.length - I;
```

The expression in such a `decreasing` clause is of an integer type. To guarantee that the variant function yields a value in a well-founded order, one technique is to prove that the expression is nonnegative. This is implied by the first loop invariant (see Figure 4). Each loop iteration decreases the value of the variant function by incrementing `I`.

Sometimes the variant function relies on the finite size of the heap and, in particular, on the finite number of objects in some chain of pointers. For example, if the specification language makes use of a hierarchical *ownership system* [Clarke et al. 1998; Leino and Müller 2004], then the *ownership relation* induced by that hierarchy forms such chains. Provided the ownership relation does not change, a specification language can allow variant functions to mention objects ordered by this well-founded order.

Recursive Procedures. Recursive procedures can encode loops and vice versa, and thus, recursive procedures have the same issues of partial correctness and termination. Loop invariants correspond to preconditions for recursive procedures since they must be true at each entry to the procedure, which corresponds to each entry to a loop. Termination is also handled similarly.

For example, JML offers the method specification clause

`measured_by` E;

which, applied to a method M, requires all calls in the implementation of M to go to methods whose value of E at the time of the call is strictly less than the value of E on entry to M. As with loops, the type of E must be some well-founded order.

Some verification tools prove termination only for some procedures. For example, Spec# proves the absence of infinite recursion only for *pure methods*, which are side-effect-free methods that can be used in assertions [Cok 2004; Leavens et al. 2005]. Because of how these pure methods are axiomatized, the soundness of the verification system depends on the lack of infinite pure-method recursion [Darvas and Müller 2006; Darvas and Leino 2007].

2.4. Exceptional Behavior

An *exception* is a way of returning from a procedure that is different than the normal return. The caller of a procedure that terminates exceptionally may handle the exception or propagate it to its caller. Exceptions pose interesting challenges for specification and verification. They introduce alternative control flow which increases the complexity of specifications and verification. Moreover, reasoning about exception handlers requires precise knowledge about the program state in which the exception was thrown, for instance, which invariants may be assumed to hold.

There are three basic approaches to dealing with exceptions. Many specification and verification techniques use different approaches depending on some categorization of the kinds of exceptions [Goodenough 1975; Leino and Schulte 2004].

(1) *Ignore.* One might choose to ignore certain kinds of exceptions entirely. For instance, when reasoning about partial correctness, one might want to ignore certain exceptions, such as stack overflow exceptions. This approach is reflected in the semantics as follows. The Hoare triple $\{P\}\,C\,\{Q\}$ is valid if and only if for any store σ that satisfies P, executing C on σ yields a store σ' that satisfies Q *or* C throws an ignored exception [Poetzsch-Heffter 1997].
(2) *Prevent.* One might choose to prevent certain kinds of exceptions from ever being thrown. For instance, array-out-of-bounds exceptions are typically program errors, which should be prevented rather than permitted and then handled. This approach is reflected in a program logic by additional proof obligations for all instructions that potentially throw an exception that is supposed to be prevented. Programmers need to write sufficiently strong preconditions and/or code to discharge these proof obligations.
(3) *Document.* One might choose to permit certain kinds of exceptions; specifications should then document under which conditions the exception may be thrown and what may be assumed about the program state when it is thrown. For instance, certain kinds of exceptions, such as I/O exceptions, are used instead of special return values to signal an unsuccessful operation and should, thus, be documented in the specification [Flanagan et al. 2002].

It is a design decision for a specification language or program verifier which of the three approaches to apply to which kinds of exceptions. JML documents exceptions that are instances of class `Exception` (e.g., `NullPointerException` and `FileNotFoundException`) and ignores all others (e.g., `VirtualMachineError`). Spec# ignores many kinds of exceptions whose prevention would be too cumbersome to specify (e.g., `StackOverflowException`); it prevents those exceptions that are typically program errors (e.g., `NullReferenceException`), and documents those that are part of the application logic (e.g., `FileNotFoundException`) [Leino and Schulte 2004]. Due to its focus on critical systems, SPARK's goal is to prevent exceptions entirely, partly through language design (for instance, absence of recursion, which makes it easier to avoid stack-overflow exceptions) and partly through proof obligations.

Choosing to ignore a possible exception leads to unsound checking, since the verifier would not analyze certain execution paths that can occur at runtime. Nevertheless, this is sometimes the practical approach to take, considering, for example, that any instruction in the .NET virtual machine can cause a `VirtualMachineError` exception.

Documenting exceptional behavior consists of describing (1) under which conditions the exception may be thrown and (2) what may be assumed about the program state when it is thrown. We illustrate these concerns with the JML example in Figure 6.

```
import java.util.EmptyStackException;

public class Stack<T> {
    Node<T> top;

    /*@  assignable top, top.prev;
      @  signals (Exception) false;
      @*/
    void push(T val) {
        top = new Node<T>(val, top);
    }

    /*@  normal_behavior
      @    requires top != null;
      @    assignable top;
      @ also
      @  exceptional_behavior
      @    requires top == null;
      @    assignable \nothing;
      @    signals (EmptyStackException) true;
      @*/
    T pop() throws EmptyStackException {
        if (top == null) throw new EmptyStackException();
        T val = top.val;
        top = top.next;
        return val;
    }
}

class Node<T> {
    T val;
    Node<T> next, prev;

    //@  assignable val, next, n.prev;
    Node(T v, Node<T> n) {
        val = v;
        next = n;
        if (n!=null) n.prev = this;
    }
}
```

Fig. 6. A JML specification of the exceptional behavior for `Stack`.

The `signals` clause in the specification of method `push` is an exceptional postcondition. It expresses that if the method throws an exception of type `Exception`, then `false` has to hold in the post-state of the method. Since this exceptional postcondition cannot be satisfied, the method must not throw such an exception. It may, however, throw an ignored exception; for instance, the creation of a new `Node` object may cause an `OutOfMemoryError`, which is not precluded by the signals clause. The specification of method `pop` contains an exceptional specification case consisting of an exceptional precondition, assignable clause, and postcondition. It expresses that `pop` may throw an exception only if `top` is null in the pre-state of the call. In this case, the method must not

modify the heap. The exceptional postcondition holds trivially. The Penelope specification language for Ada also has convenient syntax for expressing when a procedure may or must throw exceptions [Guaspari et al. 1992; Marceau 1994]. Spec# also supports exceptional pre- and postconditions. Modifies clauses in Spec# specify the locations that are potentially modified in the normal or exceptional execution; normal or exceptional postconditions can be used to strengthen the frame axiom for the respective case.

Languages with exceptions provide exception-handling constructs, such as `catch` and `finally` blocks, in Java and C#. In the presence of side effects, it is often unclear what an exception handler may assume about the state of the program, for instance, which invariants may be assumed to hold [Jacobs et al. 2007]. For documented exceptions, the exceptional frame axiom and postcondition provide this information, but for ignored exceptions, one may only assume properties that hold throughout the code that potentially throws an exception and properties the exception handler can test at runtime. So, very often, there is no safe way to recover from an ignored exception.

3. MODULES

As programs become larger, coping with their growing complexity necessitates the use of structuring techniques that aggregate subprograms and states with related functionality. For this purpose, programming languages include structuring constructs such as modules (e.g., in Modula-3), packages (e.g., in Ada), classes (e.g., in Java and C++), and aspects (e.g., in AspectJ). In our discussions, we will adopt the term *module* as a generic term for such constructs.

A crucial feature of modules is that they support *information hiding*. Essentially, this means that the module's declarations are partitioned into a public *module interface*, which is visible to all clients of the module, and a private *module body*, which contains the module implementation and is hidden from clients. The public module interface's behavior can be specified using a behavioral interface specification language. Some programming languages offer the ability to provide more than one interface to a module; for example, the `protected` mode in C++ defines an interface to a class that is visible only to those subclass clients that extend the class. Modula-3 lets the programmer define any number of interfaces to a module. A specification language may allow specification of each of these different interfaces if the specifications are distinguished by different visibility levels [Leavens and Müller 2007].

Information hiding is best enforced with a combination of programming language elements and specifications. The programming language's modules control visibility, while the specfication language gives the ability to describe behavior that is sufficient for client programming without exposing implementation details. Together, these give the provider of a module the freedom to revise implementation details as long as the implementation continues to meet its specification [Liskov and Guttag 1986; Parnas 1972].

The challenge in writing behavioral specifications for modules—and, indeed, a challenge in the design of module interface specification languages—lies in letting interface specifications say what clients need to know while avoiding exposure of implementation details. To answer this challenge, a specification language can provide data abstraction facilities that allow a module's behavior to be specified while hiding implementation details from all clients. In this section, we discuss how data can be abstracted and how one can specify invariants of module implementations. We will disscuss the related issue of how framing is done in the next section.

3.1. Data Abstraction

Data abstraction allows one to specify the behavior of a module independently of its implementation. For instance, a hash map can be specified abstractly as a partial

function from keys to values rather than in terms of its concrete representation, say, an array of linked lists of tuples of keys and values. Abstract specifications have numerous advantages over implementation-dependent specifications.

—Data abstraction can be used to specify interfaces that have no implementation, only partial implementations, or many different implementations. This expressiveness is especially needed for object-oriented programming to handle abstract classes and subtyping.
—Abstract specifications are not affected by changes of implementation details. In particular, clients that rely only on abstract specifications need not be re-verified when the implementation of a module changes.
—Abstract specifications tend to be simpler to write, read, and understand than implementation-dependent specifications because they hide certain complications such as pointers, caches, etc. In the preceding hash map example, it is much simpler to specify a look-up in terms of a partial function than to describe it in terms of arrays of lists.
—Abstract specifications are often expressed in terms of mathematical notions, such as sets, sequences, relations, etc. Many theorem provers are equipped with various lemmas and tactics for these notions, which increases automation during the verification step.
—Expressing behavior in terms of mathematical notions, such as sets and functions, allows one to formally connect the specification of a module to a high-level design specification written in languages such, as VDM, Z, or B. This approach has been pioneered by the Larch project, which developed interface specification languages that make use of mathematical vocabulary expressed in the Larch Shared Language LSL [Guttag et al. 1993].

In all approaches to data abstraction, the idea is to present a higher-level, more abstract, view of what the module implements. For instance, one higher-level view of the stack module is as a mathematical sequence. The specifications of the module's procedures can thus be given in terms of that abstract view, and the abstract view must be connected with the actual implementation.

In specification languages that are built on a specific mathematical vocabulary, such as VDM and the Larch family of interface specification languages, the abstract view is typically a value of a mathematical space, such as a sequence. In a specification language without a defined mathematical vocabulary, such as JML, the abstract view is defined within the semantics of the programming language. JML provides a library of *modeling classes* that provide a JML representation of familiar mathematical concepts. For instance, the modeling class `JMLObjectSequence` represents sequences of objects.

We explain four approaches to data abstraction: ghost variables, model variables, logic functions, and pure methods.

3.1.1. Ghost Variables. One simple approach to data abstraction is to introduce ghost variables for the abstract view in the module interface. Figure 7 shows a simple `Counter` class with JML specifications. The ghost variable `value` is used to specify the behavior of `increment` and `decrement` without exposing implementation details. Changing the internal representation of the counter, for instance, to store the value explicitly, does not affect the public specifications.

Similar to the use of ghost variables in loops (see Section 2.3.1), we can specify the connection between the ghost variables and the implementation's variables by a module invariant. Here, the invariant expresses that the counter value is the difference between the number of increments and the number of decrements. Note that this

```
public class Counter {
    private int increments;
    private int decrements;

    //@ public ghost int value;
    //@ private invariant value == increments - decrements;

    //@ public ensures value == \old(value) + 1;
    public void increment() {
        increments++;
        //@ set value = value + 1;
    }

    //@ public ensures value == \old(value) - 1;
    public void decrement() {
        decrements++;
        //@ set value = value - 1;
    }

    //@ public ensures \result == value;
    public int get() {
      return increments - decrements;
    }
}
```

Fig. 7. A JML specification of a class `Counter` using a ghost variable. Data abstraction for assignable clauses is discussed in Section 4.

invariant is not visible to clients since it refers to implementation details, namely the hidden variables `increments` and `decrements`.

To maintain this invariant, the procedure implementations in the module must update the ghost variable `value` accordingly. In JML, this is achieved by `set` statements, as illustrated in the methods `increment` and `decrement`.

The approach of using ghost variables for data abstraction has a relatively small impact on the complexity of the specification language. The biggest complication comes from the use of module invariants, where one sometimes needs to wrestle with intermediate states where the invariants do not hold (like the states between the two statements of `increment` in Figure 7). We discuss invariants further in Section 3.2.

3.1.2. Model Variables. A disadvantage of ghost variables is that they need to be updated explicitly, which may be cumbersome. *Model variables* [Cheon et al. 2005; Leino 1995], also known as *abstract variables* (in the refinement calculus), address this problem. A model variable has the appearance of a variable, but its value is defined as a function of more concrete variables, called its *representation*. Like ghost variables, model variables typically hold values of a mathematical domain, such as sets or sequences. Figure 8 shows the `Counter` example from Figure 7, this time specified using a model variable.

To a client that is aware only of the abstract view of a module, there is no interesting difference between model variables and ghost variables. However, unlike a ghost variable whose value changes only as performed by ghost-variable assignments, the value of a model variable changes immediately when there is a change in the value of a variable used in its representation. Therefore, the module body is simpler because the connection between the model variable and its representation is declared once and for all. In JML, this connection is expressed using a `represents` clause that describes how the value of the model variable can be derived from values of concrete variables.

```
public class Counter {
    private int increments;
    private int decrements;

    //@ public model int value;
    //@ private represents value = increments - decrements;

    //@ public ensures value == \old(value) + 1;
    public void increment() {
        Old: increments++;
        Here: ;
    }

    //@ public ensures value == \old(value) - 1;
    public void decrement() {
        Old: decrements++;
        Here: ;
    }

    // get method as before
}
```

Fig. 8. A JML specification of class `Counter` using a model variable. The statement labels `Old` and `Here` are used in Section 3.1.3 and can be ignored on first reading.

Like the invariant in the previous version of the example, the `represents` clause is not visible to clients.

Though simple for a user, the treatment of model variables has some complications. One complication is that the given representation function may be a partial expression. As for other partial expressions (see Section 2.1.3), one has to decide when the expression has to be defined and how to enforce that decision.

Another complication is how to deal with the far-reaching changes of model variables. When an ordinary program variable is changed, it has a potential effect on all model variables whose representation transitively depends on the variable; that is, it has a potential effect on the entire *upward closure* of the variable [Leino and Nelson 2002]. This becomes especially troublesome when model variables are fields of dynamically allocated objects and the dependencies can go through an unbounded number of objects. A solution to this complication is to let model variables occasionally go out of sync with their representation function [Leino and Müller 2006]; the semantics of model variables then reverts to the simpler semantics of ghost variables but with the convenience that the model variables are automatically updated outside the regions where they are allowed to go out of sync. This solution also suggests a policy for when representation functions need to be well defined.

3.1.3. Logic Functions. Both ghost variables and model variables typically store values of the program execution. However, it is sometime useful to abstract concrete data structures to values of a mathematical domain. This can be achieved by expressing data abstraction via mathematical functions and predicates, which (following Filliâtre and Marché [2004]) we shall refer to as *logic functions*. A logic function is defined using the expression language of the specification logic, which will typically differ from the programming language. The main difference from model variables, aside from syntax, is that logic functions are not defined using the programming language but are always defined in mathematical terms. This difference has two important consequences. First, it is clear that evaluating logic functions does not change the program state. However, the definition may depend on the program state, so changing the program state may

affect the value of the logic function. Second, the reasoning logic uses the logic function definitions directly. For example, if the logic function is defined recursively, it is up to the reasoning logic to decide how to unfold or inductively reason about the definition.

JML does not support logic functions, but the JML dialect used in the Krakatoa tool, KML, does [Marché 2009]. For our `Counter` example, we can replace the model variable `value` and its `represents` clause with the following logic function, which is declared outside class `Counter`.

```
/*@ logic int value{L}(Counter c) =
  @           \at(c.increments,L) - \at(c.decrements,L);
  @*/
```

where `L` is a label that is used to refer to a particular state and `\at` is used to refer to the state at a given label. The specification of `Counter` can refer to the abstract state of the counter by applying the above logic function, that is, by replacing `value` by `value{Here}(this)` and `\old(value)` by `value{Old}(this)` in Figure 8. Besides logic functions and their definitions, KML allows one to state axioms and lemmas which are passed directly to the theorem prover.

Traditionally, logic functions are found in all nonexecutable specification notations, like Z [Spivey 1989] and TLA [Lamport 1994]. They are also supported in many specification languages that are more closely integrated into programming languages, like in Larch [Guttag et al. 1993], ACSL [Baudin et al. 2009] and the Caduceus tool for C [Filliâtre and Marché 2004], the Krakatoa tool for Java [Marché et al. 2004; Marché 2009], VeriCool [Smans et al. 2008], and Dafny [Leino 2010]. Their use in separation logic, where they go under the name *abstract predicates* [Parkinson and Bierman 2005], is particularly interesting. There, the definitions can make use of the separating-conjunction operator, which means the abstract predicates can describe well-formedness conditions at the same time on the heap structure (see Section 4.2).

An issue with logic functions is making sure their definitions are logically consistent. This can be done using techniques like those we have already described for well-definedness of expressions (Section 2.1.3) and termination of recursive procedures (Section 2.3.2).

3.1.4. Pure Methods. While logic functions are easy to encode in program logics, they sometimes add additional specification overhead. Often classes already contain methods that express the desired abstraction, such as the `get` method in our example. Using such pure methods [Leavens et al. 2005, 2006] in specifications is the fourth and last approach to data abstraction that we consider. By declaring the method as *pure*, one indicates the intention that the method's implementation has no observable side effects on the program state, which lets assertions call the pure method. For instance, the `get` method in class `Counter` can be declared pure (see Figure 9) and then be used to specify `increment` and `decrement`.

To reason about the pure method in an assertion, one uses not the implementation of the pure method but, instead, its pre/postcondition specification. In our example, the postcondition of `get` is private because it refers to hidden implementation details. Nevertheless, this postcondition allows one to prove that `increment` and `decrement` satisfy their specifications. The implementation of a pure method which is verified to really be pure and to satisfy its specification computes a value that can be used in the dynamic checking of assertions.

Using an already existing language feature has definite appeal over introducing logic functions. In the limit, one may attempt to use pure methods as the means for users to build up algebraic theories (e.g., a theory of permutations or a theory of multisets)

```
public class Counter {
    private int increments;
    private int decrements;

    //@ public ensures get() == \old(get()) + 1;
    public void increment() {
        increments++;
    }

    //@ public ensures get() == \old(get()) - 1;
    public void decrement() {
        decrements++;
    }

    //@ private ensures \result == increments - decrements;
    /*@ pure @*/ public int get() {
      return increments - decrements;
    }
}
```

Fig. 9. A JML specification of class `Counter` using the pure method `get`.

that can be used in assertions (like the modeling classes in JML [Leavens et al. 2009]). However, the handling of pure methods has proved to be surprisingly difficult.

One problem, which is shared with logic functions, is verifying that the pure method is well defined and does not lead to an unsound axiomatization [Darvas and Müller 2006; Darvas and Leino 2007].

Another problem is checking that the method really is pure, that is, free of side effects. This is not as simple as a syntactic check for assignment statements, because one wants to allow the method body to, for example, allocate new objects (perhaps an iterator object or a string builder) and modify their state. Hence, the desired check is that of *observational purity*, which says that the method may have some side effects, but it appears pure to any observer [Barnett et al. 2004; Naumann 2007; Cok and Leavens 2008; Salcianu and Rinard 2005].

A third problem is that pure methods are not necessarily deterministic. More precisely, calling a pure method twice may yield two different results, because the side effects supposedly not visible to observers cause the second call to start in a slightly different state. This means that pure methods cannot be represented as mathematical functions of their arguments and of an unchanging heap. One solution is to make the definition of observational purity strict enough to avoid this situation; another is to allow slightly different results and to prevent callers from assuming anything more than some sort of equivalence between the results [Leino and Müller 2008].

3.1.5. Summary of Data Abstraction Techniques. All four data abstraction techniques just described allow one to express specifications without revealing implementation details. The main differences between the techniques lie in how the abstract state of a module gets updated and in whether the abstraction is expressed in the programming language or in the underlying logic.

With ghost variables, all updates of the abstract state have to be performed explicitly by the program. So for a program verifier, ghost variables behave exactly like ordinary program variables, which makes them easy to reason about and trivial to support in specification languages and verification tools. However, the explicit updates sometimes lead to a large specification overhead, especially for recursive data structures. In contrast, model variables, logic functions, and pure methods automatically reflect changes

in the concrete representation without any explicit updates. A downside of these implicit changes is that framing—that is, deciding which abstract values are affected by an update of a concrete variable—becomes challenging and requires other verification techniques, which we discuss in Section 4.

Logic functions are defined as mathematical functions in the program logic, which gives them a precise semantics and avoids the problems of checking purity and determinism. However, logic functions require programmers to be familiar with two different formalisms: the programming language and the mathematical notation in which the logic functions are defined. In contrast, the values assigned to ghost variables, the representations of model variables, and the specifications and implementations of pure methods are typically expressed in the programming language. While this leads to the complications just discussed, it simplifies runtime assertion checking and facilitates reuse. For instance, modules typically provide various methods for observing the state of the module, and these methods can also be used in specifications as pure methods. Finally, model variables can be seen as parameter-less pure methods. The absence of parameters greatly simplifies framing, see Section 4.

3.2. Module Invariants

Data abstraction offers a way to talk about the module's state in the module interface and to connect that abstract view with the module body's state. But proving the correctness of a module body also relies on knowing that its concrete data structures are in a good state [Hoare 1972], for instance, that certain variables hold non-null value or that an index lies within the bounds of an array. Such properties about data structures are, in general, captured by *module invariants*.

Module invariants might be categorized as being private or public. A *private* module invariant is visible only to the module body, and hence it can depend only on variables (e.g., fields) declared in the module body. A *public* module invariant can mention public variables and other entities declared in the module interface but cannot depend on private variables. Such public invariants let clients of the module know about properties that are ensured by the module, for example, that the stack size lies within some bound. Making certain invariants private preserves information hiding and avoids the reverification of clients when implementation details and the corresponding invariants change [Leavens and Müller 2007].

When module invariants mention more than one variable, it becomes necessary to allow some program points where an invariant is temporarily violated. A question is then, when does a module invariant hold? A simple answer is that a module invariant holds whenever control is outside the module. The simple answer suffices in the idyllic sequential setting where a module invariant constrains the state of only one module and where the entry and exit points of the module can be determined statically. A module invariant is then checked at the end of the module initialization and at each module exit point; in return, the module invariant can be assumed to hold at each module entry point.

However, these assumptions do not apply to realistic programs. First, it is common to use invariants to relate the state of several modules. For instance, such *multimodule invariants* relate the objects in a data structure (e.g., the subject and its observers in the Observer pattern [Gamma et al. 1995]) or constrain the components of an aggregate structure (e.g., the array used to implement a list). A special case of multimodule invariants occurs with inheritance in object-oriented programs, where subclass invariants often constrain the values of fields that are inherited from superclasses, that is, declared in a different module. Second, the entry and exit points of a module are often not statically known, for instance, in the presence of procedure pointers or dynamically dispatched methods. A complication, then, is the possibility of *callbacks*, that is, when

a module calls a procedure while the module invariant is temporarily violated, the procedure might call back and reenter the module under the false assumption that the module invariant holds. With procedure pointers and dynamic binding, such callbacks cannot, in general, be detected with complete precision by a modular static analysis. In the following, we summarize solutions to these two issues and glance at alternatives to module invariants.

3.2.1. Multimodule Invariants. When invariants constrain the state of multiple modules, then the update of a single variable potentially violates numerous module invariants. For instance, an update of a subject module might temporarily violate the invariants of all its observers until the observers are notified and their invariants get reestablished. The key issue in reasoning about multimodule invariants is how to enforce statically and modularly that all invariants that are potentially violated by an update get reestablished. There are three major solutions.

(1) *Encapsulation*. A variable is encapsulated in a structure (such as a module or an object) if it can be assigned to only by procedures of that structure. When an invariant is restricted to constrain only encapsulated variables, then one can impose proof obligations on the procedures of the structure to enforce that the invariant is preserved [Müller et al. 2006]. A typical example is aggregate structures, where the subcomponents of the aggregate are modified only through the aggregate component.

(2) *Visibility*. Even when invariants depend on variables that are not encapsulated, one can determine the necessary checks statically and modularly if an invariant is visible at all variable assignments that possibly violate the invariant [Müller et al. 2006]. This visibility requirement is found in recursive data structures (e.g., the invariant that relates two nodes of a doubly linked list) and when several cooperating modules are declared in a larger context (such as a package or assembly), for instance, a list and its iterator.

(3) *Monotonicity*. Invariants may depend on variables of arbitrary modules if the variables evolve monotonically and the invariant is preserved by the monotonic updates. For instance, if a counter variable is only incremented, then an invariant stating that the counter is positive is never violated after it has once been established. The two most common forms of monotonicity are initialization (a variable goes from the uninitialized state to the initialized state but never back) [Fähndrich and Xia 2007] and immutability (a variable is not updated at all after its initialization) [Leino et al. 2008]. More general forms of monotonicity are explored in type states [Fähndrich and Leino 2003; Pilkiewicz and Pottier 2009], object relations [Leino and Schulte 2007; Cohen et al. 2010], and concurrency [Jones 1983; Cohen et al. 2010].

3.2.2. Callbacks. A common semantics for module invariants is to require module invariants to hold in all *visible states*, that is, in the pre- and post-states of all procedures [Drossopoulou et al. 2008; Meyer 1997]. To avoid that a callback enters a module under the false assumption that the module invariant holds, techniques based on a visible state semantics enforce that each module invariant is reestablished before calling a procedure that potentially calls back. Since it is typically not known statically which procedures call back, invariants need to be reestablished before each call, which is often inconvenient and may be impossible if the call itself is necessary for reestablishing the invariant.

An alternative to the visible state semantics is to specify explicitly in each procedure's precondition which invariants it requires to hold [Leino and Müller 2004]. It is then possible to make calls while an invariant is violated, as long as the called procedure does not require that invariant to hold. If it does not, the procedure might still call back but not under the false assumption that the invariant holds. This approach, which is implemented in Spec#, is more flexible than visible state semantics but requires good

default specifications and abstraction mechanisms to keep the specification overhead low.

3.2.3. Alternatives. Specification and verification techniques that handle the invariants occurring in modern (especially object-oriented) programs in a sound and modular way are nontrivial. An alternative to module invariants is to specify the required constraints explicitly in pre- and postconditions. The data abstraction mechanisms described in Section 3.1 can be used to preserve information hiding and to express constraints on multiple modules. This approach solves the callback problem trivially because there are no implicit assumptions when certain conditions hold. Dealing with multimodule invariants is reduced to framing model variables, logic functions, or pure methods, which we discuss in Section 4. Replacing invariants by logic functions (abstract predicates) is common in separation logic [Parkinson and Bierman 2005]. ESC/Modula-3 [Leino and Nelson 2002] explored this specification pattern using a model variable `valid` to express that an object is in a good state.

4. OBJECTS AND THE HEAP

In order to specify the behavior of objects and other heap data structures, a specification technique must be able to describe (a) the topology of the data structures and (b) the effects that methods have on them. We illustrate these two facets using a Java class `Node` with fields `a` and `b` of type `Node`.

A specification of the *topology* of a data structure typically answers the following questions.

—What is the shape of the implemented data structure? For example, instances of `Node` could represent a doubly-linked list (with `a` and `b` pointing to the predecessor and successor node, respectively) or a binary tree (with `a` and `b` pointing to the left and right child node, respectively). Information about the shape is typically needed to prove functional correctness of a data structure, for instance, deletion of a node works differently for lists and trees.
—Does the data structure contain cycles? For example, our nodes could form a cyclic or an acyclic list. Information about cyclicity is, for instance, needed to program and prove termination or deadlock-freedom of list traversals.
—Which objects of a data structure are potentially shared? For example, for a tree structure, we want to express that the left and right subtrees are disjoint. Similarly, we might want to express that two instances of a class `List` do not share any nodes. Information about sharing is needed to reason about the effects of modifications, for instance, to prove that modifying the nodes of one list does not affect any other list.

A specification of the *effects* of the methods of a data structure may include the following.

—Write effects. Which parts of the heap are potentially modified by a method? This information is needed to determine which properties of the heap are affected by a method.
—Read effects. Which parts of the heap are potentially read by a method? This information is needed to reason about dependencies, both for knowing when a pure method's value may change (see Section 3.1.4) and for reasoning about interference in a concurrent program.
—Allocation and deallocation effects. Which objects are allocated or deallocated by a method? Information about allocation is, for instance, needed to prove that the result

of a method is different from all existing objects. Deallocation information is needed to verify programs with explicit memory management.

—Locking information. Which locks are acquired or released by a method? This information is needed to prove the absence of data races and deadlocks.

In order to be useful for program verification, specification techniques for topologies and effects must address three major challenges.

(1) Abstraction. Heap properties have to be expressed in an implementation-independent way. Abstraction is important for preserving information hiding and to support subtyping [Leavens and Müller 2007; Leino 1998]. For instance, a write effect specification that simply lists the (concrete) fields modified by a method breaks information hiding by revealing field names does not work for interfaces and abstract classes, which might not have fields and is too restrictive for subclasses, which might have to modify additional fields.

(2) Reasoning. The formal framework in which heap properties are expressed should allow efficient, ideally automatic reasoning. For instance, topology properties are often expressed using reachability predicates, which are notoriously difficult to handle for automatic theorem provers, such as SMT solvers.

(3) Framing. One of the most important reasoning steps that any specification of heap operations must support is framing, that is, proving that certain heap properties are not affected by a heap operation [Gorelick 1975]. The following invariance rule illustrates framing.

$$\frac{\{P\}\,C\,\{Q\}}{\{P \wedge R\}\,C\,\{Q \wedge R\}} \text{ if the write effect of } C \text{ is disjoint from } \mathrm{FV}(R)$$

This rule suffices if R is a predicate that only involves simple variables and $\mathrm{FV}(R)$ denotes the free variables of R. The rule says that an assertion about some state that is unmodified by C can be preserved. However, with complex heap data structures, the side condition becomes more troublesome: the disjointness of the effects of C and the assertion R becomes more difficult to specify. We need to be able to specify that the *footprint* of C—the locations that C changes—is disjoint from the footprint of R—the locations that R depends upon.

In the following, we present three major approaches to the specification of heap properties. The first approach uses *explicit footprints*, that is, sets of objects (or locations) that are used in predicates and effect specifications. The second approach uses *implicit footprints* which are derived from predicates in specialized logics. The third approach uses *predefined footprints* that are derived from predefined heap topologies. We illustrate the three approaches with an example of an unbounded stack that is implemented using a linked list. We present its implementation in Figure 10. This implementation uses the `Node` class from Figure 6 to represent the stack as a doubly-linked list.

4.1. Explicit Footprints

The basic idea of this approach is to represent the footprint of data structures and predicates directly in the logic as sets of objects or sets of locations. We will call these sets *regions* in the following. A specification typically contains (a) a specification of regions which can be expressed as ghost variables or as functions of the heap, (b) predicates that describe the topology of a data structure and other well-formedness conditions, and (c) read and write effects for the operations of the data structure, expressed in terms of the specified regions.

The explicit footprints approach was pioneered in Dynamic Frames [Kassios 2006], which uses heap-dependent logic functions (see Section 3.1.3) to describe regions. In the

```
public class UnboundedStack<T> {
    private Node<T> top;

    public void push(T val) {
        top = new Node<T>(val,top);
    }

    public T pop() {
        T val = top.val;
        top = top.next;
        return val;
    }

    public boolean isEmpty() {
        return top==null;
    }
}
```

Fig. 10. A Java implementation of an unbounded stack. Class `Node` is shown in Figure 6.

```
function Valid(): bool
  reads {this} + footprint;
{
  this in footprint && !(null in footprint) &&
  (next == null ==> content == [val])  &&
  (next != null ==> next in footprint && next.footprint < footprint &&
                    !(this in next.footprint) && next.prev == this &&
                    content == [val] + next.content && next.Valid())
}
```

Fig. 11. A logic function defining a list Node's footprint in Dafny.

following example, we describe how dynamic frames, as found in the Dafny language [Leino 2010] express aspects (a)-(c).

(a) We specify the footprint of the doubly-linked list of `Node` objects using a (ghost) field:

```
ghost var footprint: set<Node<T>>;
```

which has to be updated explicitly in the code whenever the footprint of the list changes, for instance, when a node is added. (b) The contents of `footprint`, as well as other well-formedness conditions, are specified by the boolean function shown in Figure 11, where `+` is overloaded to denote set union and sequence concatenation, `in` denotes set membership, `<` is overloaded to denote the strict subset operator, `{_}` constructs a singleton set, and `[_]` constructs a singleton sequence.

The function `Valid` expresses that `footprint` contains at least the current node `this` and its successors. It also specifies the topology. Here `next.prev == this` ensures that each node is correctly linked to its successor. Since the footprint of the successor must be a subset of the footprint of the current node but not contain the current node, we can conclude that the list is acyclic. The ghost field `content` ranges over sequences of `T`; it is used to specify the functional behavior of the list. (c) The region `footprint` is used to specify the read effect of `Valid`.

Figure 12 shows the Dafny version of class `UnboundedStack`. Like `Node`, it declares a field to store the footprint. Function `IsUnboundedStack` specifies the

```
class UnboundedStack<T> {
  var top: Node<T>;
  ghost var footprint: set<object>;
  ghost var content: seq<T>;

  function IsUnboundedStack(): bool
    reads {this} + footprint;
  {
    this in footprint &&
    (top == null  ==>  content == []) &&
    (top != null  ==>  top in footprint &&
                       top.footprint <= footprint &&
                       top.prev == null && content == top.content &&
                       top.Valid())
  }

  function IsEmpty(): bool
    reads {this};
  { content == [] }

  method InitUnboundedStack()
    modifies {this};
    ensures IsUnboundedStack() && IsEmpty();
  { top := null;  footprint := {this};  content := []; }

  method Push(val: T)
    requires IsUnboundedStack();
    modifies footprint;
    ensures IsUnboundedStack() && content == [val] + old(content);
  {
    var tmp := new Node<T>;  call tmp.InitNode(val,top);
    top := tmp;  footprint := footprint + {tmp};
    content := [val] + content;
  }

  method Pop() returns (result: T)
    requires IsUnboundedStack() && !IsEmpty();
    modifies footprint;
    ensures IsUnboundedStack() && content == old(content)[1..];
  {
    result := top.val;
    assert top.Valid();
    footprint := footprint - top.footprint;  top := top.next;
    if (top != null) {
      footprint := footprint + top.footprint;  top.prev := null;
    }
    content := content[1..];
} }
```

Fig. 12. A Dafny version of `UnboundedStack`. Dafny does not have constructors; therefore, method `InitUnboundedStack` is used to initialize a new instance. We implemented `IsEmpty` as a function instead of a method to be able to use it in specifications.

well-formedness conditions for stacks, especially that the footprint of a stack contains the footprint of its nodes and that the nodes are well formed. Methods `Push` and `Pop` illustrate how regions are used to specify write effects using a modifies clause.

The explicit footprint technique addresses the three challenges just described as follows.

(1) Abstraction. Effects are specified in terms of regions which do not reveal implementation details, such as names of fields. The content of a region is typically underspecified, which allows subclasses to extend inherited regions. For instance, function `IsUnboundedStack` only requires that at least the stack and the footprint of its first node be in the footprint, but subclasses might add more objects if needed.

To specify footprints implementation-independently, one can apply any of the data abstraction techniques outlined in Section 3.1. As initially presented by Kassios, dynamic frames use logic functions to describe regions. VeriCool [Smans et al. 2008b] uses pure methods for this purpose. When using an automatic theorem prover to reason about footprints, recursively defined pure methods or logic functions can be a hampering issue due to the possibility that the prover will instantiate function definitions indefinitely. The problem can be curbed instead by using ghost variables, as in Dafny [Leino 2010] and region logic [Banerjee et al. 2008].

Data groups [Leino 1998; Leino et al. 2002] can also be used to specify footprints within a single object. They are very similar to model variables (see Section 3.1.2). A data group represents a region. Typically, a data group is declared in a module interface (like a model variable); the contents of the data group are declared in the module body (like the representation of the model variable).

(2) Reasoning. The original work on dynamic frames [Kassios 2006] uses higher-order logic, which is difficult to automate. However, subsequent work (for instance on VeriCool, Dafny, and region logic), developed dynamic frames in a first-order setting. Zee et al. [2008], have used explicit footprints successfully to verify the functional correctness of linked data structures in Jahob. Banerjee et al. [2008] report on promising experiments with an encoding of region logic in the intermediate verification language Boogie [Leino and Rümmer 2010]. However, the VCC project [Cohen et al. 2009] experienced performance problems with dynamic frames, which made them switch to a flexible ownership system instead.

(3) Framing. With explicit footprints, framing is done by proving that the read effect of a predicate and the write effect of a method are disjoint. For instance, for well-formed stacks `s1` and `s2`, the following code verifies in Dafny (where `!!` denotes disjointness of sets).

```
assume s1.footprint !! s2.footprint;
assume s1.IsEmpty();
call s2.Push(o);
assert s1.IsEmpty();
```

The call to `Push` preserves `s1.IsEmpty()` because the read effect of `s1.IsEmpty()` is `{s1}`. Since `s1` is well formed, we know that `s1` is in `s1.footprint` and thus disjoint from `s2.footprint`, which is the write effect of the call to `s2.Push`.

4.2. Implicit Footprints

Next we consider specialized logics, which allow for elegant implicit representations of the footprints. The two main representatives of this approach are separation logic [O'Hearn et al. 2001, 2004; Reynolds 2000, 2002] and implicit dynamic frames [Smans et al. 2008a; Leino and Müller 2009]. In this section, we focus on separation logic, which extends the assertion language of Hoare logic with a new connective: the separating conjunction $P * Q$. Each assertion in separation logic defines a portion of the heap. $P * Q$ means that the current portion of the heap can be split into two disjoint portions such that one satisfies P and the other Q.

The key to separation logic is local reasoning, that is, a specification needs to describe all the state that the code uses (even if it just reads from it). The triple $\{P\}\,C\{Q\}$ means that all the state the code C needs to execute is described by P. That is, P describes precisely the footprint of the code C. This interpretation of triples leads to the following key rule of separation logic.

$$\frac{\{P\}\,C\,\{Q\}}{\{P * R\}\,C\,\{Q * R\}}$$

This rule allows any proof to be extended by disjoint state and to know that this disjoint state is unmodified. This is valid as P must describe all the heap that C uses, and hence, it cannot change any additional state.

Note that some versions of separation logic require side conditions about local variable modification, as $*$ only separates heaps. Separation logic can be extended such that $*$ is used to also separate variables, so that these side conditions are not required [Parkinson et al. 2006].

In the rest of this section, we use a separation logic for Java based on Parkinson's work [Parkinson 2005; Parkinson and Bierman 2005]. This separation logic has a primitive assertion for describing the value of a field in the heap, $x.f \mapsto v$. It means the heap contains at least field f of the object x, which has value v. Using $*$ we can describe the portion of the heap containing several disjoint objects.

Separation logic uses predicates (logic functions, see Section 3.1.3) to specify data structures. We define a predicate `Node` to specify the state associated with a `Node` object.

$$\texttt{Node}(x, v, n, p) \quad = \quad x.\texttt{val} \mapsto v \;*\; x.\texttt{next} \mapsto n \;*\; x.\texttt{prev} \mapsto p$$

This simply states that we have three fields for node x: `val`, `next`, and `prev`—the first pointing to v, the second to n, and the third to p. The predicate $\texttt{Node}(x, v, n, p)$ implicitly defines a footprint consisting of $x.\texttt{val}$, $x.\texttt{next}$, and $x.\texttt{prev}$.

Using this predicate, we can define a recursive predicate `List` for a doubly-linked list.

$$\begin{aligned} \texttt{List}(x, \texttt{nil}(), p) \quad &= \quad x = \texttt{null} \\ \texttt{List}(x, \texttt{cons}(v, vs), p) \quad &= \quad \exists n.\ \texttt{Node}(x, v, n, p) \;*\; \texttt{List}(n, vs, x) \end{aligned}$$

The first argument to this predicate indicates the head of the linked list, the second specifies the contents of the list as a sequence, and the final specifies the value of the previous pointer coming from the doubly-linked list. The definition is split into two cases: (a) empty $\texttt{nil}()$, the head is simply `null`; and (b) nonempty $\texttt{cons}(v, vs)$, the list is composed of a node with value v and the rest of the list with contents vs. Importantly, for the second case, the node and the rest of the list are disjoint. This means the list cannot be cyclic. The implicit footprint defined by $\texttt{List}(x, cons(v, vs), p)$ contains the `val`, `next`, and `prev` fields of all nodes reachable from x via `next` references.

Finally, we package the list into a predicate that represents the `Stack` class.

$$\texttt{Stack}(s, vs) \quad = \quad \exists l, p.\ s.\texttt{top} \mapsto l \;*\; \texttt{List}(l, vs, p)$$

This has a single field `top` that points to the start of the list. Its implicit footprint contains $s.\texttt{top}$ as well as all locations in the footprint of the underlying list.

We use this predicate to specify the operations on the stack in Figure 13, in the syntax of jStar specification files (each Java method signature is followed by two formula—the first, the precondition and the second, the post). Note that this specification does not contain explicit read or write effects (modifies clauses). They are implicit in the method preconditions. By mentioning the `Stack` predicate in the precondition of a method, the method gains the permission to read and write all locations in the footprint

```
class UnboundedStack {
   UnboundedStack() :
   {}
   { Stack(this, nil()) };

   void push(Object val) :
   { Stack(this, vs) }
   { Stack(this, cons(val,vs)) };

   Object pop() :
   { Stack(this, cons(v,vs)) }
   { Stack(this, vs) * v == return};

   boolean isEmpty() :
   { Stack(this, vs) }
   { Stack(this, vs)
      *  (return==false() * vs == cons(_,_)
           || return==true() * vs == nil() ) };
}
```

Fig. 13. A separation logic specification for class `UnboundedStack`. The variables `v` and `vs` are logical variables, as described in Section 2.2.4. The _ represents anonymous existentials, each occurence can have a different value.

of the predicate; the method cannot access any memory outside of this footprint. To compensate for the absence of explicit write effect specifications, method `isEmpty` needs a postcondition that states that the method will preserve the stack.

Returning to the three challenge properties for specifying topologies and effects, we have the following.

(1) Abstraction. Separation logic and implicit dynamic frames achieve abstraction through logic functions. The specification of the stack does not reveal implementation details, as they are packaged up in the definition of the predicate. The second parameter to the stack predicate represents the interesting data stored in the stack. The definitions of the predicates relate this value to the actual memory layout of the data. If this predicate is scoped so that the client does not know the definition, then the definition is *abstract*, and hence, the client cannot depend on any internal structure of the stack [Parkinson and Bierman 2005]. We could give exactly the same specification if the stack were implemented using a singly or doubly-linked list, or an array that is dynamically resized when full.

(2) Reasoning. There have been many hand proofs of separation logic programs. There is now a series of tools that can reason using separation logic, including Smallfoot [Berdine et al. 2005], SpaceInvader [Distefano et al. 2006], SLAyer[7], jStar [Distefano and Parkinson 2008], and VeriFast [Jacobs and Piessens 2008]. Automatic reasoning about separation logic formulae is an active area of research but still lags behind classical logic.

Implicit dynamic frames can be encoded in first-order logic and are, thus, well suited to automatic verification with SMT solvers, as illustrated by VeriCool [Smans et al. 2008b] and Chalice [Leino et al. 2009].

(3) Framing. Framing with implicit footprints is practically trivial. In the specification, we have not had to specify modifies clauses or footprints. They are implicit in the definition of the predicate. The predicates describe the shape of the stack. The code

[7]`http://research.microsoft.com/en-us/um/cambridge/projects/slayer/`.

cannot access any memory outside of this shape. As $*$ expresses disjointness, we simply have to describe the precondition of the method as a separate area of the heap, and then whatever heap is not in the area needed by the precondition will be preserved by the call. Consider the following.

```
assume Stack(s1, xs) * Stack(s2, ys);
s2.push(o);
assert Stack(s1, xs) * Stack(s2, cons(o,ys));
```

Here, the call to `push` only needs to find the stack predicate for `s2`. It is automatic that all the state disjoint from that predicate, that is, other predicates composed with $*$, will be preserved.

4.3. Predefined Footprints

Whereas the first two approaches describe the heap properties found in a program, the third approach is to enable efficient reasoning for programs with restricted topologies. We will discuss ownership [Clarke et al. 1998] as a representative for this approach.

Most ownership systems enforce a tree topology: every object in the heap has at most one owner object, and the owner relation is acyclic. Topological properties beyond this tree structure have to be expressed using object invariants and predicate logic. Read and write effects typically use ownership as an abstraction mechanism: the right to read or write an object includes the right to read or write all the objects it (transitively) owns.

We illustrate this approach using Spec#. Spec# expresses ownership via attributes on field declarations. For instance, the `[Peer]` attribute in the following field declarations in class `Node` express that a `Node` object has the same owner as its successor and its predecessor.

```
[Peer] Node<T> next;
[Peer] Node<T> prev;
```

Additional topological properties, such as the fact that the nodes form a list and that the list is acyclic, have to be expressed via object invariants.

```
invariant next != null ==> next.prev == this;
invariant prev != null ==> prev.next == this;
invariant 0 < len;
invariant len == (next == null ? 1 : next.len + 1);
```

The integer field `len` is needed to specify acyclicity of the list formed by the nodes. Class `UnboundedStack` is shown in Figure 14. Its `top` field is declared with the `[Rep]` attribute, which expresses that the stack owns its first node and, since all nodes in the list are peers, all nodes. Therefore, different stacks are guaranteed to have disjoint node structures. When a field declaration has no ownership attribute, its owner is not specified and can be arbitrary. This is the case for the objects stored on a stack.

Spec# uses modifies clauses to specify write effects. Essentially, the modifies clause of method `Push` expresses that the method may modify all fields of `this` (denoted by the wildcard, $*$), as well as the fields of all objects (transitively) owned by `this`. This interpretation allows the method to change the underlying node structure without mentioning these implementation details in the modifies clause.

Read effects are expressed using the `Reads` attribute. The default for a pure method is that it may read the state of its receiver and all objects owned by the receiver. This default applies, for instance, to method `IsEmpty` in our example. Again, ownership is used to abstract from the internal representation of the stack.

```
public class UnboundedStack<T> {
  [Rep] private Node<T> top;
  invariant top != null ==> top.prev == null;

  sealed model int height {
    satisfies height == (top == null ? 0 : top.len);
  }

  public void Push(T val)
    modifies this.*;
    ensures height == old(height) + 1;
  {
    Node<T> tmp = new Node<T>(val);
    expose(this) {
      if (top == null) { tmp.len = 1; }
      else             { top.Prepend(tmp); }
      top = tmp;
    }
  }

  [return: NoDefaultContract] public T Pop()
    requires height > 0;
    modifies this.*;
    ensures height == old(height) - 1;
  {
    assert top != null;  // follows from precondition
    Node<T> r = top;
    expose(this) {
      top = top.next;
      if (top != null) {
        expose(top) {
          expose(r) { r.next = null;  r.len = 1; }
          top.prev = null;
        }
      }
    }
    return r.val;
  }

  [Pure] public bool IsEmpty()
    ensures result == (height == 0);
  { return top == null; }
}
```

Fig. 14. A Spec# version of `UnboundedStack`. The `satisfies` clause in the declaration of the model field `height` provides the representation of `height`. An `expose(o)` block indicates program points at which `o`'s object invariant may be broken. The attribute `[return: NoDefaultContract]` on method `Pop` disables a default postcondition that requires the result's invariant to hold, a property would require an ownership relation between the stack and its elements. Method `IsEmpty` is declared pure such that it can be called in specifications.

The three challenges are addressed as follows.

(1) Abstraction. The effect specifications of `UnboundedStack` do not reveal implementation details. The wildcard allows one to abstract over field names, whereas ownership is used to abstract over the owned objects, which form the internal representation of the stack.

(2) Reasoning. Ownership has been used to verify invariants [Drossopoulou et al. 2008; Leino and Müller 2004; Müller et al. 2006] and write effects [Müller et al. 2003]. All the existing ownership-based verification techniques enforce that all modifications of an object must be initiated by its owner. This strong encapsulation property gives owners full control over modifications of their internal representation and, thus, allows them to maintain invariants. Ownership has been shown to be useful also in reasoning about model fields [Leino and Müller 2006] and in enforcing object immutability [Leino et al. 2008].

The ownership topology and encapsulation can be enforced by type systems [Lu et al. 2007; Müller 2002], for instance, through Universe Types [Dietl and Müller 2005] as is the case in JML, or through logic-based reasoning like in Spec# [Leino and Müller 2004]. In the latter case, ownership annotations are encoded as object invariants and then verified in a program logic.

(3) Framing. Since ownership enforces a tree structure on the heap, we know that the ownership trees rooted in two distinct objects o_1 and o_2 are disjoint if neither o_1 owns o_2 nor vice versa. The disjointness of ownership trees can then be used to prove that read and write effects of methods do not overlap. For instance, in the following code snippet, we can prove that the call to `s2.Push` does not affect the result of `s1.IsEmpty` because the write effect of the former is the ownership tree rooted in `s2` and the read effect of the latter in the tree rooted in `s1`. Since `s1` and `s2` are distinct objects with the same owner, these trees are known to be disjoint.

```
assume s1 != s2 && Owner.Same(s1, s2);
assume s1.IsEmpty();
s2.Push(o);
assert s1.IsEmpty();
```

4.4. Summary of Heap Specification Techniques

All three techniques previously described allow one to specify the topologies of many common object structures and the effects of their operations. They have been used as the basis of several program verifiers. However, the three techniques strike a different balance between expressiveness and automation.

Techniques based on explicit footprints are very expressive, since they allow specifications to relate different regions in arbitrary ways, for instance, to express the disjointness or inclusion of regions or to characterize their intersection. Therefore, these techniques can easily specify overlapping regions (sharing) or cyclic structures. However, this flexibility complicates reasoning. When regions are stored explicitly in ghost variables, programs need to update these ghost variables explicitly to maintain invariants. Especially for recursive data structures, writing these updates can be cumbersome. Alternatively, when regions are computed by pure methods or logic functions, one needs to reason explicitly about the effects of heap modifications on the results of these functions, that is, on the contents of the regions.

Techniques based on implicit footprints are almost as expressive as techniques with explicit footprints, but any sharing or cycles must be made explicitely, which can complicate the reasoning. Verifiers based on separation logic have mostly applied symbolic execution and not yet achieved the same level of automation as verifiers based on verification condition generation. Recent work on implicit dynamic frames in Chalice

[Leino and Müller 2009] showed how to reason automatically about implicit footprints using verification conditions and automatic theorem provers. Investigating whether this approach is also applicable to separation logic is ongoing research.

Finally, techniques based on predefined footprints are very convenient to use for data structures whose topology is supported by the technique, but they provide no support for other structures. For instance, specification techniques based on ownership handle hierarchic aggregate structures with very little specification overhead but fail to specify non-hierarchical topologies, such as the Observer pattern. The predefined topologies allow program verifiers to apply specialized verification strategies and, thus, increase automation. For instance, a verifier can exploit the hierarchy of ownership topologies to show the termination of traversals of such structures.

5. SUBTYPING AND INHERITANCE

Object-oriented programming (OOP) presents many challenging problems for specification and verification. We have already discussed issues related to abstract data type specification and heap manipulation, both of which are prominent features of OOP. The other essential characteristic of OOP is the use of subtyping and dynamic dispatch.

In type systems, *subtype polymorphism* allows variables and expressions to denote values of several different but related types at runtime; for example, a variable `coll` of static type `Collection` might denote an object of some `Collection` subtype, such as `Stack`, `Set`, or `Bag`. Subtyping in this sense is a purely type-theoretic property that requires that each instance of a subtype can be manipulated as if it were an instance of its supertypes without type errors. For example, if the type system allows a method call such as `coll.add(e)`, then since `coll` denotes an object of some subtype of `Collection`, the call must not produce a runtime type error. Cardelli [1988] published an influential study of the conditions for type checking OOP. Cardelli's subtyping rules prevent type errors and can be extended to languages with more features, such as multiple dispatch [Castagna 1995]. This use of subtype polymorphism is at the core of most object-oriented design patterns [Gamma et al. 1995].

Dynamic dispatch is a semantic feature of OOP languages that allows a method call, such as `coll.add(e)`, to have different effects depending on the runtime type of the receiver object, `coll`. The programming language dynamically determines what implementation to run for such a call based on the runtime type of the receiver. The selected implementation may be provided by `coll`'s static type, `Collection`, or instead of being inherited in this way, it may be overridden by a method in a subtype of `Collection`. So in general, the executed implementation might be one of any number of different implementations, some of which might not have been imagined when the call was written.

However, it is not only the implementation to be executed that is not known statically (in general): without some methodological convention, such as behavioral subtyping, even the specification of the method that will be executed will not be known statically. An example is the code shown in Figure 15, which assumes that the size of the `Collection` `coll` is zero and that the object `e` is a legal element of the collection; it then calls `coll`'s `add` method, which is dynamically dispatched. Because the specification of the code executed may vary among subtypes, this kind of code presents a problem for static verification. One approach to solving this problem is to verify what a method call does for each possible subtype of the receiver's type [von Oheimb 2001]. However, such a case analysis would be difficult to maintain, as each time a new subtype were added to the program, the case analysis code would have to be extended.

A more modular approach is to follow the analogy of object-oriented type systems and impose restrictions on the behavior of subtypes: this methodology is known as

```
//@ assignable coll.*;
public void testAdd(Collection coll, Object e) {
    //@ assume coll.size() == 0 && coll.legalElement(e);
    boolean b = coll.add(e);
    //@ assert coll.contains(e);
    //@ assert b ==> coll.size() == 1;
}
```

Fig. 15. A test of `Collection`'s `add` method.

```
public interface Collection {

    //@ model instance JMLDataGroup state;

    /*@ pure @*/ int size();

    /*@ pure @*/ boolean legalElement(Object o);

    /*@ pure @*/ boolean contains(Object o);

    /*@ public behavior
      @   requires legalElement(o);
      @   assignable state;
      @   ensures contains(o);
      @   ensures \result ==> size() == \old(size()+1);
      @   ensures !\result ==> size() == \old(size());   @*/
    boolean add(Object o);
}
```

Fig. 16. A JML specification for `Collection`.

behavioral subtyping [America 1987; Dhara and Leavens 1996; Leavens 2006; Liskov 1988; Liskov and Wing 1994; Meyer 1997]. Using behavioral subtyping, one can use the specification of a supertype's objects to soundly reason about the behavior of all possible subtype objects.[8] For example, behavioral subtyping would allow one to use the specification of `Collection` to draw conclusions about the call to `add` in Figure 15. Behavioral subtyping validates such conclusions because the contract for `Collection`'s `add` method, given in Figure 16, says that the element is contained in the collection after the call, and when the result of the call is true, then the collection's size has been increased by 1. Put the other way around, with behavioral subtyping, subtype objects must behave according to the instance specification of each of their supertypes when they are manipulated using that supertype's interface [Liskov 1988]. For example, the `add` method of a `Collection` subtype must obey the contract given in Figure 16. Just as a type system with subtyping guarantees that no runtime type errors occur when calling methods using dynamic dispatch, verifying behavioral subtyping ensures that no surprising behavior occurs when calling methods using dynamic dispatch [Leavens and Weihl 1995].

[8]An alternative to behavioral subtyping is class refinement [Back et al. 2000], which requires that, in addition to behavioral subtyping, all constructors (and class/static methods) of a class refine the corresponding constructors of their supertypes. This suffices for sound reasoning, but is stronger than needed if the client code that one reasons about only manipulates existing objects and does not call constructors.

A simple way of enforcing behavioral subtyping is to impose the following four rules for every supertype C and subtype D [America 1987; Liskov and Wing 1994].

(1) For each instance method $D.m$ that overrides a method $C.m$, the precondition of $C.m$ must imply the precondition of $D.m$. That is, overriding methods may weaken preconditions.
(2) For each instance method $D.m$ that overrides a method $C.m$, the set of locations that $D.m$ may modify must be a subset of the set of modifiable locations for $C.m$. That is, overriding methods may strengthen frame conditions.
(3) For each instance method $D.m$ that overrides a method $C.m$, the postcondition of $D.m$ must imply the postcondition of $C.m$. That is, overriding methods may strengthen postconditions.
(4) The invariant for D objects must imply the invariant for C objects. That is, subtypes may strengthen invariants.

Rules (1)–(3) allow one to reason about a call to $o.m$ using the specification for m in o's static type C. By rule (1), establishing the precondition of $C.m$ before the call guarantees that the precondition of the method selected at runtime, $D.m$, also holds. By rule (2), $D.m$ can only modify locations that $C.m$'s specification allows to be changed; thus, one can safely conclude that if $C.m$'s specification guarantees that the value of some location x is left unchanged, then $D.m$ does as well. By rule (3), one may assume $C.m$'s postcondition after the call, because $D.m$ establishes a postcondition that is at least as strong.

Rule (4) is necessary for handling inheritance. A method implementation $C.m$ may assume C's invariant. By rule (4), this assumption is still justified when m is inherited by a subtype D. However, there can be problems if a method implementation $C.m$ is called on a D object, since $C.m$ will not know how to establish the potentially stronger invariant of D objects. A drastic solution to this problem is to outlaw invariants that mention variables that are declared in a superclass [Müller et al. 2006]. Another way around this problem is to require $C.m$ to be overridden in cases where it can modify the object's state in ways that threaten the invariant of D [Ruby and Leavens 2000]. Yet another solution is to re-verify all inherited and non-overridden methods [Parkinson and Bierman 2008] in subclass D. Finally, the Spec# methodology solves this problem by tying each invariant to a class and carefully restricting when fields can be modified and when each class's invariant must be reestablished [Barnett et al. 2004; Leino and Müller 2004; Leino and Wallenburg 2008]. Rule (4) also applies to other consistency criteria of objects, such as history constraints [Liskov and Wing 1994].

One way to summarize these rules is to say that all overriding subtype methods must satisfy the specification of each method that they override, which is necessary for sound modular reasoning using a supertype's method specification [Dhara and Leavens 1996; Leavens and Weihl 1995].

However, the programming language and the verification logic guarantee additional properties for dynamically dispatched calls, which can be used to weaken these rules and still maintain soundness [Chen and Cheng 2000]. Weaker rules are beneficial, since they allow more types to be behavioral subtypes and give developers more freedom in design and implementation of subtypes.

The programming language guarantees that an overriding method in class D will be called only when the receiver's class is D (or a subtype of D). In specification frameworks where the invariant of the receiver has to hold in the pre- and post-state of a call, this implies that one may assume the D-invariant of the receiver to hold (which, according to rule (4) may be stronger than the C-invariant). These type and invariant guarantees allow for weaker versions of rules (1)–(3). For each instance method $D.m$ that overrides a method $C.m$, the following hold.

(D1) The precondition of $C.m$ must imply the precondition of $D.m$, provided that the receiver is of type D and the D-invariant of the receiver holds.
(D2) The frame of $D.m$ must be a subset of the frame condition of $C.m$, provided that the receiver is of type D and the D-invariant of the receiver holds in the pre- and post-state.
(D3) The postcondition of $D.m$ must imply the postcondition of $C.m$, provided that the receiver is of type D and the D-invariant of the receiver holds in the pre- and post-state.

For instance, rule (D3) allows $D.m$ to have a weaker postcondition than $C.m$ if this weaker postcondition, together with the D-invariant, implies the postcondition of $C.m$.

The knowledge that the receiver is of class D is not only useful for assuming D's invariant, but for all specification elements that may be refined in subclasses, such as model fields [Leino 1995; Leino and Müller 2006; Müller 2002] and pure methods [Darvas and Müller 2005]. For instance, the possible values of a model field may be restricted in subclasses. Thus, more precise type information for the receiver provides more information about the values of its model fields.

Verification logics guarantee that methods are called only in states in which their preconditions hold. This enables a weaker version of the frame and postcondition rules, which only require that a supertype's frame and postcondition are obeyed when that supertype's precondition is also held in the call's pre-state [Dhara and Leavens 1996; Chen and Cheng 2000].

(D2′) The frame of $D.m$ must be a subset of the frame condition of $C.m$, provided that the receiver is of type D, the D-invariant of the receiver holds in the pre- and post-state, and $C.m$'s precondition holds in the pre-state .
(D3′) The postcondition of $D.m$ must imply the postcondition of $C.m$, provided that the receiver is of type D, the D-invariant of the receiver holds in the pre- and post-state, and $C.m$'s precondition holds in the pre-state.

To simplify the application of the rules for behavioral subtyping, many specification languages use *specification inheritance* [Dhara and Leavens 1996; Leavens 2006]. Subtypes inherit the specifications of their supertypes and can refine the inherited specifications by adding declarations. A simple way to define the *effective specification* of a class is to combine the inherited and the declared specification as follows:[9] the *effective precondition* of a method $D.m$ is the disjunction of the precondition declared for $D.m$ and the preconditions of the methods it overrides. Taking the disjunction guarantees that the effective precondition is weaker than the preconditions of all overridden methods and, thus, that rule (1) is satisfied. The *effective frame* of $D.m$ is the intersection of the frame declared for $D.m$ and the frame of the methods it overrides. Using this intersection guarantees that the effective frame is a subset of the frames of the methods it overrides and, thus, that rule (2) is satisfied. The *effective postcondition* of $D.m$ is the conjunction of the postcondition declared for $D.m$ and the postconditions of the methods it overrides.[10] Taking the conjunction guarantees that the effective postcondition is stronger than the postconditions of all overridden methods and, thus, that rule (3) is satisfied. Finally, the *effective invariant* of a class D is the conjunction of the invariant declared for D and the invariants of D's supertypes. Taking the conjunction guarantees

[9]This simple form of specification inheritance is based on the behavioral subtyping rules (1)–(4) [Liskov and Wing 1994]. Eiffel [Meyer 1997] also has a simple form of specification inheritance with similar rules but has no frame conditions.
[10]This rule for forming the effective postcondition is stronger than necessary to satisfy condition (D3′) [Dhara and Leavens 1996] and, indeed, JML uses a more relaxed rule [Leavens 2006].

```
public class ArrayList implements Collection {
    private /*@ spec_public @*/ Object[] elements =
                                new Object[3]; //@ in state;
    //@ maps elements[*] \into state;
    private /*@ spec_public @*/ int siz = 0; //@ in state;

    //@ invariant elements != null && elements.length > 0;
    //@ invariant 0 <= siz && siz <= elements.length;

    //@ also
    //@   ensures \result == siz;
    public /*@ pure @*/ int size() { return siz; }

    //@ also
    //@   ensures \result;
    /*@ pure @*/ public boolean legalElement(Object o) {
        return true;
    }

    /*@ also
      @  ensures \result == (\exists int i; 0 <= i && i < siz;
      @                                  elements[i] == o); @*/
    public /*@ pure @*/ boolean contains(Object o) {
        for (int i = 0; i < siz; i++) {
            if (elements[i] == o) { return true; }
        }
        return false;
    }

    //@ also
    //@   requires true;
    //@   assignable elements, elements[siz], siz;
    //@   ensures \result;
    public boolean add(Object o) {
        if (siz == elements.length) {
            Object[] tmp = new Object[siz*2];
            System.arraycopy(elements,0,tmp,0,siz);
            elements = tmp;
        }
        elements[siz++] = o;
        return true;
    }
}
```

Fig. 17. A JML specification for `ArrayList`, which is a behavioral subtype of `Collection`. JML uses the `also` keyword to highlight the fact that a method specification in a subtype will be combined with an inherited specification.

that the effective invariant is stronger than the invariants of D's supertypes and, thus, that rule (4) is satisfied.

An example illustrating these rules is shown in Figure 17. This subtype of `Collection` is a behavioral subtype because it obeys the specification of `Collection`. We illustrate how specification inheritance works using method `add`. The method satisfies rule (1)

by providing the requires clause `true`; thus, the effective precondition is the disjunction of `Collection`'s precondition with `true` (which is equivalent to true), and thus weaker than `Collection`'s precondition. The method satisfies rule (2) because the effective frame is the intersection of the frames given by `Collection`'s `add` method and `ArrayList`'s (in fact, according to the data group declarations in `Arraylist`, `ArrayList`'s is a subset of the assignable locations listed in `Collection`). Method `add` satisfies rule (3) by providing the ensures clause `\result`; thus, the effective postcondition is equivalent to `contains(o) && size() == \old(size()+1) && \result`, which is stronger than `Collection`'s postcondition. Finally, `ArrayList` satisfies rule (4) by contributing additional conjuncts to the effective invariant.

Specification inheritance enforces behavioral subtyping [Dhara and Leavens 1996; Leavens and Naumann 2006]. However, the preceding rules have been criticized as disguising specification errors, because failure to comply with the rules of behavioral subtyping has silently turned into unsatisfiable specifications. Assume that an overridden supertype method has the declared precondition `p > 0` and that the overriding subtype method requires `p <= 0`. With specification inheritance, however, the effective precondition of the overriding method is equivalent to `true`. Findler and Felleisen [2001] argue that this should be considered a specification error which should be reported, since the disjunction of these preconditions will silently accept calls that violate either the supertype's precondition or the subtype's. (In JML and other languages that use specification inheritance, the subtype's method would have to accept all these calls and satisfy both specifications [Leavens 2006].)

Despite this criticism, most existing specification languages enforce behavioral subtyping through specification inheritance. JML [Leavens 2006; Leavens et al. 2009] defines effective preconditions and invariants as previously described. For postconditions, JML exploits behavioral subtyping rule (D3′): in the effective postcondition, the postcondition of a method $C.m$ only has to hold if the corresponding precondition held before the call. That is, the effective postcondition is a conjunction of implications of the form $\bigwedge_c(\texttt{old}(pre_c) \Rightarrow post_c)$, rather than a conjunction of postconditions $\bigwedge_c post_c$. The rule for frame properties similarly depends on which precondition holds. This is the meaning behind the JML `also` keyword.

Eiffel's rules are very similar to JML's, but in the effective postcondition of a method $D.m$, the declared postcondition of $D.m$ has to hold, even if the corresponding precondition did not hold before the call [Eiffel 2005]. The resulting effective postcondition has the form $\bigwedge_c(\texttt{old}(pre_c) \Rightarrow post_c) \wedge post_D$, which is stronger than necessary for soundness. Spec# [Barnett et al. 2005] uses a stricter rule for preconditions. Overriding methods must not change the inherited precondition. When a method implements signatures from more than one supertype (interface), the preconditions in these supertypes must be identical. The effective postcondition of a method is simply the conjunction of all inherited and declared postconditions. Thus, with Spec#'s precondition rule, a distinction among different precondition cases is not useful. Finally, in Spec#, an overriding method must not declare additional modifies clauses. A larger modifies clause would be unsound with Spec#'s precondition rule, and a smaller modifies clause, that is, strengthening of the frame properties, can be achieved through additional postconditions.

Even though behavioral subtyping is an important design principle for the safe use of subtyping and inheritance, there are implementations where a subclass restricts or changes the behavior of its superclass. To support such implementations, it has been proposed to distinguish between static and dynamic method specifications [Chin et al. 2008; Parkinson and Bierman 2008; Poetzsch-Heffter and Müller 1999]. A *static specification* describes the behavior of a particular method implementation, whereas a *dynamic specification* describes the common behavior of all implementations of a

method, including overriding methods in subclasses. In other words, static specifications are not inherited by subtypes and are not subject to behavioral subtyping. Hence, static specifications are used to reason about statically bound calls where the implementation to be executed is known statically—for instance, Java's `super` calls or calls to `final` methods; dynamic specifications are used to reason about dynamically dispatched calls.

6. CONCLUSIONS

Specifying the behavior of complex programs requires highly expressive notations. For instance, the functional specification of a compiler includes a specification of the syntax of the source and target language, of the semantics of both languages, and of the translation function [Leroy 2006]. Mathematical logics, such as higher-order logic, provide the required expressiveness but are difficult to automate. Proofs in these frameworks are largely interactive, and verifying programs beyond toy examples still takes heroic efforts by highly specialized experts.

It is one of the main objectives of specification language designers to find idioms that strike a good trade-off between expressiveness, usability by programmers, and automated verification. In this article, we surveyed behavioral interface specification languages for sequential programs and presented some specification language features that are good compromises between these competing goals.

(1) They are sufficiently expressive to specify the functional correctness of program components and complex data structures (see, for instance, the composite pattern specifications presented at the 2008 ACM Specification and Verification of Component-Based Systems Workshop).
(2) They are based on relatively simple concepts, which can be taught to undergraduate students and applied by programmers (see, for instance, an industrial JML case study [Cataño and Huisman 2002]).
(3) They have been used successfully in automatic program verifiers, such as ESC/Java, Spec#, Jahob, or jStar (see, for instance, the automatic verification of linked-list structures in Jahob [Zee et al. 2008]).

Despite these successes, there are a number of research challenges that need to be addressed in order to achieve the objectives of the Verified Software Initiative [Hoare 2003]. The following are some of the key challenges.

(1) *Extending the well-established specification techniques for sequential programs to concurrency*. Even though the foundations for reasoning about concurrent programs are available (for instance, temporal logic to express liveness properties [Lamport 1994], rely-guarantee reasoning to specify shared-variable concurrency [Xu et al. 1997], or permissions to specify thread synchronization [Boyland 2003]), the development of specification languages that can express functional behavior of programs using advanced concurrency primitives is in an early stage and not ready to be applied to mainstream programs [Jacobs et al. 2008; O'Hearn 2007; Leino and Müller 2009; Rodríguez et al. 2005], with the notable exception of VCC [Cohen et al. 2010], which is being applied to Microsoft's Hyper-V virtualization platform, a large concurrent C-program.

(2) *Linking detailed design specifications to requirement specifications, analysis-level specifications, and software architecture*. Behavioral interface specifications languages can express properties of software components, but expressing the interaction between components is still a challenge. For instance, although there are techniques for specifying the Observer pattern [Banerjee et al. 2008], reasoning about the correctness of an event-driven system seems to be beyond the state-of-the-art technigues. Another such example is data-driven architectures, where the program is essentially an interpreter of some data, such as business rules, which encode the actual application logic.

Verifying such applications will require a formal connection between architectural properties and detailed design specifications.

(3) *Increasing the automation of verification*. Today's automatic verifiers are still limited to fairly simple properties, such as the absence of exceptions. Even though steady progress is being made [Zee et al. 2008; Leino 2010], the verification of the functional correctness of whole applications still requires significant user interaction. This challenge will partly be addressed by the theorem-proving community, but it will also be necessary to improve programming, specification, and verification methodology in order to produce verification conditions that are more easily accessible to automatic theorem provers, such as SMT solvers [Barrett and Tinelli 2007; de Moura and Bjørner 2008].

Addressing these and other challenges in ways that are useful for the Verified Software Initiative will require close collaboration of experts in programming and specification language design, verification methodology, and theorem proving. It is, thus, important to foster collaborations among these communities through joint projects, tool integration, as well as challenges and competitions.

ACKNOWLEDGMENTS

Thanks to Tony Hoare and Jaydev Misra for their persistent encouragement to write this survey and to the referees for their reading and comments. Thanks also to Claude Marché for his explanation of logic functions.

REFERENCES

Abadi, M. and Lamport, L. 1988. The existence of refinement mappings. Tech. rep. 29, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA.

Abrial, J.-R. 1996. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, Cambridge, U.K.

Aldrich, J., Chambers, C., and Notkin, D. 2002. Architectural reasoning in ArchJava. In *Proceedings of the 16th European Conference, Object-Oriented Programming (ECOOP'02)*, B. Magnusson, Ed. Lecture Notes in Computer Science, vol. 2374. Springer-Verlag, Berlin, 334–367.

Alhir, S. S. 1998. *UML in a Nutshell*. O'Reilly, Sebastapol, CA.

Alur, R., Courcoubetts, C., and Dill, D. 1990. Model checking for real-time systems. In *Proceedings of the 5th Annual Sympostum on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, CA, 414–425.

Alur, R. and Henzinger, T. A. 1994. A really temporal logic. *J. ACM 41,* 1, 181–203.

Ambler, A. L., Good, D. I., Browne, J. C., Burger, W. F., Choen, R. M., Hoch, C. G., and Wells, R. E. 1977. Gypsy: A language for specification and implementation of verifiable programs. In *Proceedings of the ACM Conference on Language Design for Reliable Software*, SIGPLAN Not. *12,* 3, 1–10.

America, P. 1987. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'87)*, J. Bezivin et al., Eds. Lecture Notes in Computer Science, volume 276, Springer-Verlag, New York, NY, 234–242.

Andrews, D. J. 1996. Information technology programming languages, their environments and system software interfaces: Vienna Development Method–specification language – part 1: Base language. International Standard ISO/IEC 13817-1, International Standards Organization.

Apt, K. R. 1981. Ten years of Hoare's logic: A survey—part I. *ACM Trans. Program. Lang. Syst. 3,* 4, 431–483.

Apt, K. R. and 0lderog, E. 1991. Introduction to program verification. In *Formal Description of Programming Concepts*, E. J. Neuhold and M. Paul, Eds, IFIP State-of-the-Art Reports. Springer-Verlag, New York, NY, 363–429.

Apt, K. R. and Olderog, E. 1997. *Verification of Sequential and Concurrent Programs, 2nd ed.* Graduate Texts in Computer Science Series. Springer-Verlag, New York, NY.

Arlow, J. and Neustadt, I. 2005. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design 2nd ed.* Addison-Wesley, Indianoplis, IN.

Back, R.-J., Mikhajlova, A., and von Wright, J. 2000. Class refinement as semantics of correct object substitutability. *Formal Aspects Comput. 12,* 1, 18–40.

BACK, R. J. R. 1980. *Correctness Preserving Program Refinements: Proof Theory and Applications*. Mathematical Center Tracts, vol. 131. Mathematical Centre, Amsterdam.

BACKHOUSE, R., CHISHOLM, P., MALCOLM, G., AND SAAMAN, E. 1989. Do-it-yourself type theory. *Formal Aspects Comput. 1,* 1, 19–84.

BALL, T. AND RAJAMANI, S. K. 2001. The SLAM toolkit. In *Computer Aided Verification*. Lecture Notes in Computer Science, vol. 2102. Springer-Verlag, Berlin, 260–264.

BANERJEE, A., BARNETT, M., AND NAUMANN, D. A. 2008. Boogie meets regions: A verification experience report. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, N. Shankar and J. Woodcock, Eds. Lecture Notes in Computer Science, vol. 5295. Springer-Verlag, New York, NY, 177–191.

BANERJEE, A., NAUMANN, D. A., AND ROSENBERG, S. 2008. Regional logic for local reasoning about global invariants. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, J. Vitek, Ed. Lecture Notes in Computer Science, vol. 5142. Springer-Verlag, New York, NY, 387–411.

BARNES, J. 1997. *High Integrity Ada: The SPARK Approach*. Addison Wesley Longman, Inc., Reading, MA.

BARNETT, M., CHANG, B.-Y. E., DELINE, R., JACOBS, B., AND LEINO, K. R. M. 2006. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*. Lecture Notes in Computer Science, vol. 4111. Springer-Verlag, New York, NY, 364–387.

BARNETT, M., DELINE, R., FÄHNDRICH, M., LEINO, K. R. M., AND SCHULTE, W. 2004. Verification of object-oriented programs with invariants. *J. Object Technol. 3,* 6, 27–56.

BARNETT, M., LEINO, K. R. M., AND SCHULTE, W. 2005. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds. Lecture Notes in Computer Science, vol. 3362. Springer-Verlag, New York, NY, 49–69.

BARNETT, M., SCHULTE, D. A. N. W., AND SUN, Q. 2004. 99.44% pure: Useful abstractions in specification. In *Formal Techniques for Java-like Programs (FTfJP)*, E. Poll, Ed. Radbod University, Nijmegen, 11–19. `http://www.cs.ru.nl/ftfjp/2004/Purity.pdf`.

BARNETT, M. AND SCHULTE, W. 2003. Runtime verification of .NET contracts. *J. Syst. Softw. 65,* 3, 199–208.

BARRETT, C. AND TINELLI, C. 2007. CVC3. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, W. Damm and H. Hermanns, Eds. Lecture Notes in Computer Science, vol. 4590. Springer-Verlag, Berlin, 298–302.

BARRINGER, H., CHENG, J. H., AND JONES, C. B. 1984. A logic covering undefinedness in program proofs. *Acta Informatica 21,* 3, 251–269.

BARTETZKO, D., FISCHER, C., MOLLER, M., AND WEHRHEIM, H. 2001. Jass - Java with assertions. In *Proceedings of the Workshop on Runtime Verification held in conjunction with the 13th Conference on Computer Aided Verification (CAV'01)*. Published in *Electronic Notes in Theoretical Computer Science*, K. Havelund and G. Rosu, Eds., *55*, 2.

BAUDIN, P., FILLIÂTRE, J.-C., MARCHÉ, C., MONATE, B., MOY, Y., AND PREVOSTO, V. 2009. *ACSL: ANSI/ISO C Specification Language, version 1.4*. ANSI. `http://frama-c.cea.fr/acsl.html`.

BECKERT, B., HÄHNLE, R., AND SCHMITT, P. H. 2007. *Verification of Object-Oriented Software: The KeY Approach*. Lecture Notes in Computer Science, vol. 4334. Springer-Verlag, Berlin.

BERDINE, J., CALCAGNO, C., AND O'HEARN, P. W. 2005. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects (FMCO)*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds. Lecture Notes in Computer Science, vol. 4111. Springer-Verlag, Berlin, 115–137.

BERRY, G. 2000. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. The MIT Press, Cambridge, MA.

BIDOIT, M., KREOWSKI, H.-J., LESCANNE, P., OREJAS, F., AND SANNELLA, D., Eds. 1991. *Algebraic System Specification and Development: A Survey and Annotated Bibliography*. Lecture Notes in Computer Science, vol. 501. Springer-Verlag, Berlin.

BJØRNER, D. AND HENSON, M. C. 2008. *Logics of Specification Languages*. Springer-Verlag, Berlin.

BLANCHET, B., COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. 2002. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation*, T. A. Mogensen, D. A. Schmidt, and I. H. Sudborough, Eds. Lecture Notes in Computer Science, vol. 2566. Springer-Verlag, 85–108.

BÖRGER, E. AND STÄRK, R. 2003. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, Berlin.

BORGIDA, A., MYLOPOULOS, J., AND REITER, R. 1995. On the frame problem in procedure specifications. *IEEE Trans. Softw. Eng. 21,* 10, 785–798.

BOYAPATI, C., KHURSHID, S., AND MARINOV, D. 2002. Korat: Automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 123–133.

BOYLAND, J. 2003. Checking interference with fractional permissions. In *Static Analysis (SAS)*, R. Cousot, Ed. Lecture Notes in Computer Science, vol. 2694. Springer-Verlag, Berlin, 55–72.

BROOKES, S. D., HOARE, C. A. R., AND ROSCOE, A. W. 1984. A theory of communicating sequential processes. *J. ACM 31,* 3, 560–599.

BURDY, L., CHEON, Y., COK, D. R., ERNST, M. D., KINIRY, J. R., LEAVENS, G. T., LEINO, K. R. M., AND POLL, E. 2005. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transfer 7,* 3, 212–232.

CARDELLI, L. 1988. A semantics of multiple inheritance. *Inform. Comput. 76,* 2–3, 138–164.

CARTWRIGHT, R. 1981. Formal program testing. In *Proceedings of the Conference Record of the 8th ACM Symposium on Principles of Programming Languages*. 125–132.

CASTAGNA, G. 1995. Covariance and contravariance: Conflict without a cause. *ACM Trans. Program. Lang. Syst. 17,* 3, 431–447.

CATAÑO, N. AND HUISMAN, M. 2002. Formal specification of Gemplus's electronic purse case study. In *FME 2002*: *Formal Methods-Getting IT Right*, L. H. Eriksson and P. A. Lindsay, Eds. Lecture Notes in Computer Science vol., 2391. Springer-Verlag, Berlin, 272–289.

CHALIN, P. 2005. Logical foundations of program assertions: What do practitioners want? In *Proceedings of the 3rd International Conference on Software Engineering and Formal Method (SEFM)*. IEEE Computer Society, Los Alamitos, CA.

CHALIN, P. 2007. A sound assertion semantics for the dependable systems evoluation verifying compiler. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, Los Alamitos, CA, 23–33.

CHANG, J. AND RICHARDSON, D. J. 1999. Structural specification-based testing: Automated support and experimental evaluation. In *Software Engineering – ESEC/FSE '99*, O. Nierstrasz and M. Lemoine, Eds. Lecture Notes in Computer Science, vol. 1687. Springer-Verlag, 285–302. Also *ACM SIGSOFT Softw. Engineer. Notes*, *24*, 6.

CHAPMAN, R. 2000. Industrial experience with SPARK. *ACM SIGADA Ada Letters 20,* 4, 64–68.

CHEN, Y. AND CHENG, B. H. C. 2000. A semantic foundation for specification matching. In *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, New York, NY, 91–109.

CHEON, Y. AND LEAVENS, G. T. 1994. The Larch/Smalltalk interface specification language. *ACM Trans. Softw. Eng. Methodol. 3,* 3, 221–253.

CHEON, Y. AND LEAVENS, G. T. 2002. A runtime assertion checker for the Java Modeling Language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02)*, H. R. Arabnia and Y. Mun, Eds. CSREA Press, 322–328.

CHEON, Y. AND LEAVENS, G. T. 2005. A contextual interpretation of undefinedness for runtime assertion checking. In *Proceedings of the 6th International Symposium on Automated and Analysis-Driven Debugging, (AADEBUG'05)*. ACM Press, New York, NY, 149–157.

CHEON, Y., LEAVENS, G. T., SITARAMAN, M., AND EDWARDS, S. 2005. Model variables: Cleanly supporting abstraction in design by contract. *Soft. Pract. Exp. 35,* 6, 583–599.

CHIN, W.-N., DAVID, C., NGUYEN, H. H., AND QIN, S. 2008. Enhancing modular OO verification with separation logic. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, P. Wadler, Ed. ACM, New York, NY, 87–99.

CLARKE, D. G., POTTER, J. M., AND NOBLE, J. 1998. Ownership types for flexible alias protection. In *Proceedings of the ACM Annual Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'98), SIGPLAN Not.*, *33*, 10. ACM, New York, NY, 48–64.

CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. 1986. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8,* 2, 244–263.

CLARKE, M. J., GRUMBERG, O., AND PELED, D. A. 1999. *Model Checking*. The MIT Press, Cambridge, MA.

CLARKE, L. A. AND ROSENBLUM, D. S. 2006. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Softw. Eng. Notes 31,* 3, 25–37. `http://doi.acm.org/10.1145/1127878.1127900`.

COFI (THE COMMON FRAMEWORK INITIATIVE). 2004. CASL *Reference Manual*. Lecture Notes in Computer Science, vol. 2960 (IFIP Series). Springer-Verlag, Berlin.

COHEN, E., DAHLWEID, M., HILLEBRAND, M., LEINENBACH, D., MOSKAL, M., SANTEN, T., SCHULTE, W., AND TOBIES, S. 2009. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics,*

22nd International Conference, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Lecture Notes in Computer Science, vol. 5674. Springer-Verlag, Berlin, 23–42.

COHEN, E., MOSKAL, M., TOBIES, S., AND SCHULTE, W. 2010. Local verification of global invariants in concurrent programs. In *Computer Aided Verification (CAV)*. Lecture Notes in Computer Science. Springer-Verlag, Berlin. Forthcoming.

COK, D. AND LEAVENS, G. T. 2008. Extensions of the theory of observational purity and a practical design for JML. In *Proceedings of the 7th International Workshop on Specification and Verification of Component-Based Systems (SAVCBS'08)*. Tech. Rep. CS-TR-08-07. School of EECS, University of Central Florida, Orlando, FL, 43–50.

COK, D. R. 2004. Reasoning with specifications containing method calls in JML and first-order provers. In *Proceedings of the ECOOP Workshop FTfJP'2004 Formal Techniques for Java-like Programs*, E. Poll, Ed. Rabound University, Nijmegen, 41–48.

CONSTABLE, R. L., ALLEN, S., H. BROMELY, CLEVELAND, W., ET AL. 1986. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, Inc., Englewood Cliffs, NJ.

CORBETT, J. C., DWYER, M. B., HATCLIFF, J., LAUBACH, S., PASAREANU, C. S., ROBBY, AND ZHENG, H. 2000. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*. ACM Press, New York, NY, 439–448.

COUSOT, P. 1990. Methods and logics for proving programs. In *Handbook of Theoretical Computer Science*, J. van Leewen, Ed. Vol. B: Formal Models and Semantics. The MIT Press, New York, NY, 841–993.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 238–252.

CRISTIAN, F. 1984. Correct and robust programs. *IEEE Trans. Softw. Engi. 10*, 163–174.

DARVAS, Á. AND LEINO, K. R. M. 2007. Practical reasoning about invocations and implementations of pure methods. In *Fundamental Approaches to Software Engineering, 10th International Conference*, A. L. Matthew B. Dwyer, Ed. Lecture Notes in Computer Science, vol. 4422. Springer-Verlag, Berlin, 336–351.

DARVAS, Á. AND MÜLLER, P. 2005. Reasoning about method calls in JML specifications. In *Formal Techniques for Java-like Programs*. ETH, Zurich, Switzerland.

DARVAS, Á. AND MÜLLER, P. 2006. Reasoning about method calls in interface specifications. *J. Object Technol. 5,* 5, 59–85.

DE MOURA, L. AND BJØRNER, N. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis (TACAS)*. Lecture Notes in Computer Science, vol. 4963. Springer-Verlag, Berlin, 337–340.

DENG, X., LEE, J., AND ROBBY. 2006. Bogor/Kiasan: A *k*-bounded symbolic execution for checking strong heap properties of open systems. In *Proceedings of the 21st IEEE Conference on Automated Software Engineering*. 157–166.

DETLEFS, D. L., LEINO, K. R. M., NELSON, G., AND SAXE, J. B. 1998. Extended static checking. SRC Research rep. 159, Compaq Systems Research Center, Palo Alto, CA.

DHARA, K. K. AND LEAVENS, G. T. 1996. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 258–267. A corrected version is ISU CS TR #95-20c, `http://tinyurl.com/s2krg`.

DIETL, W. AND MÜLLER, P. 2005. Universes: Lightweight ownership for JML. *J. Object Technol. 4,* 8, 5–32.

DIJKSTRA, E. W. 1976. *A Discipline of Programming*. Prentice-Hall, Inc., Englewood Cliffs, NJ.

DISTEFANO, D., O'HEARN, P. W., AND YANG, H. 2006. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 3920. Springer-Verlag, Berlin, 287–302.

DISTEFANO, D. AND PARKINSON, M. J. 2008. jStar: Towards practical verification for Java. In *ACM Annual Conference of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, G. E. Harris, Ed. ACM, New York, NY, 213–226.

DONG, J. S., HAO, P., ZHANG, X., AND QIN, S. C. 2006. Highspec: A tool for building and checking OZTA models. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*. ACM, New York, NY, 775–778.

DROSSOPOULOU, S., FRANCALANZA, A., MÜLLER, P., AND SUMMERS, A. J. 2008. A unified framework for verification techniques for object invariants. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, J. Vitek, Ed. Lecture Notes in Computer Science. Springer-Verlag, Berlin, 412–437.

EDWARDS, S. H., HEYM, W. D., LONG, T. J., SITARAMAN, M., AND WEIDE, B. W. 1994. Part II: Specifying components in RESOLVE. *ACM SIGSOFT Soft. Engi. Notes 19,* 4, 29–39.

EIFFEL. 2005. Eiffel analysis, design and programming language. ECMA Standard 367.

EMERSON, E. A. 1990. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. B. The MIT Press, Cambridge, MA, 995–1072.

ERNST, M. D. 2003. Static and dynamic analysis: Synergy and duality. In *Proceedings of the ICSE Workshop on Dynamic Analysis, (WODA'03)*. New Mexico State University, NM, 24–27.

FÄHNDRICH, M. AND LEINO, K. R. M. 2003. Heap monotonic typestates. In *Proceedings of the ECOOP International Workshop on Aliasing, Confinement and Ownership (IWACO'03)*. K. U. Leuven, Leuven.

FÄHNDRICH, M. AND XIA, S. 2007. Establishing object invariants with delayed types. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. Steele Jr., Eds. ACM, New York, NY, 337–350.

FILLIÂTRE, J.-C. AND MARCHÉ, C. 2004. Multi-prover verification of C programs. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'04)*. Lecture Notes in Computer Science, vol. 3308. Springer-Verlag, Berlin, 15–29.

FINDLER, R. B. AND FELLEISEN, M. 2001. Contract soundness for object-oriented languages. In *Proceedings of the Annual ACM SIGPLAN Conference of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*. ACM, New York, NY, 1–15.

FINDLER, R. B. AND FELLEISEN, M. 2002. Contracts for higher-order functions. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, NY, 48–59.

FITZGERALD, J. 2008. The typed logic of partial functions and the Vienna Development Method. In *Logics of Specification Languages,* Springer-Verlag, Berlin, 453–487.

FITZGERALD, J. AND LARSEN, P. G. 1998. *Modelling Systems: Practical Tools in Software Development*. Cambridge Press, Cambridge, U.K.

FITZGERALD, J. S., LARSEN, P. G., MUKHERJEE, P., PLAT, N., AND VERHOEF, M. 2005. *Validated Designs for Object-Oriented Systems*. Springer-Verlag, London.

FLANAGAN, C. 2006. Hybrid type checking. In *Proceedings of the 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*. *SIGPLAN Not. 41*, 1, ACM Press, New York, 245–256.

FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. 2002. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. *SIGPLAN Not. 37*, 5. ACM, New York, NY, 234–245.

FLOYD, R. W. 1967. Assigning meanings to programs. *Proc. Symp. Appl. Math. 19*, 19–31.

FRANCEZ, N. 1992. *Program Verification*. Addison-Wesley Publishing Co., Cambridge, U.K.

FURIA, C. A., MANDRIOLI, D., MORZENTI, A., AND ROSSI, M. 2010. Modeling time in computing: A taxonomy and a comparative survey. *ACM Comput. Surv. 42,* 2, 1–59.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

GARLAN, D., MONROE, R. T., AND WILE, D. 2000. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, New York, NY, 47–67.

GOGUEN, J. A., THATCHER, J. W., AND WAGNER, E. G. 1978. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology*, vol. 4, R. T. Yeh, Ed. V Prentice-Hall, Inc., Englewood Cliffs, NJ., 80–149.

GOODENOUGH, J. B. 1975. Exception handling: Issues and a proposed notation. *Commun. ACM 18,* 12, 683–696.

GORELICK, G. A. 1975. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Tech. rep. 75, University of Toronto, Toronto, Canada.

GRIES, D. AND SCHNEIDER, F. B. 1995. Avoiding the undefined by underspecification. In *Computer Science Today: Recent Trends and Developments*, J. van Leeuwen, Ed. Lecture Notes in Computer Science, no. 1000. Springer-Verlag, New York, NY, 366–373.

GRIESKAMP, W., TILLMANN, N., AND SCHULTE, W. 2005. XRT - exploring runtime for .NET - architecture and applications. In *Proceedings of the Workshop on Software Model Checking*.

GUASPARI, D., MARCEAU, C., AND POLAK, W. 1992. Formal verification of Ada programs. In *Proceedings of the 1st International Workshop on Larch*, U. Martin and J. M. Wing, Eds. Springer-Verlag, New York, NY, 104–141.

GUREVICH, Y. 1991. Evolving algebras: A tutorial introduction. *Bullet. EATCS 43*, 264–284.

GUTTAG, J. V. AND HORNING, J. J. 1986. Report on the Larch Shared Language. *Sci. Comput. Program. 6,* 2, 103–134.

GUTTAG, J. V., HORNING, J. J., GARLAND, S. J., JONES, K. D., MODET, A., AND WING, J. M. 1993. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY.

GUTTAG, J. V., HORNING, J. J., AND WING, J. M. 1985. The Larch family of specification languages. *IEEE Software 2,* 5, 24–36.

HALBWACHS, N., LAGNIER, F., AND RATEL, C. 1992. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. Softw. Eng. 18,* 9, 785–793.

HANSEN, M. R. 2008. Duration calculus. In *Logics of Specification Languages*, Springer-Verlag, Berlin, 299–347.

HAREL, D. 1987. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program. 8,* 3, 231–274.

HAREL, D., LACHOVER, H., NAAMAD, A., PNUELI, A., POLITI, M., SHERMAN, R., SHTULL-TRAURING, A., AND TRAKHTENBROT, M. 1990. Statemate: A working environment for the development of complex reactive systems. *IEEE Trans. Softw. Eng. 16,* 4, 403–413.

HAYES, I., Ed. 1993. *Specification Case Studies*, 2nd ed. International Series in Computer Science. Prentice-Hall, Inc., London, U.K.

HEHNER, E. C. R. 1989. Termination is timing. In *Mathematics of Program Construction*, J. L. A. van de Snepscheut, Ed. Lecture Notes in Computer Science, vol. 375. Springer-Verlag, Berlin, 36–47.

HEHNER, E. C. R. 1993. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, New York, NY. `http://www.cs.utoronto.ca/~hehner/aPToP`.

HEHNER, E. C. R. 1998. Formalization of time and space. *Formal Aspects Comput. 10*, 290–306.

HEHNER, E. C. R. 2005. Specified blocks. Verified Software: Theories, Tools, Experiments (VSTTE). `http://tinyurl.com/2a7kf2`.

HEIMDAHL, M. P. E., WHALEN, M. W., AND THOMPSON, J. M. 2003. NIMBUS: A tool for specification centered development. In *Proceedings of the 11th IEEE International Conference on Requirements Engineering (RE'03)*. IEEE Computer Society, Los Alamitos, CA, 349.

HEITMEYER, C., JEFFORDS, R., BHARADWAJ, R., AND ARCHER, M. 2007. Re theory meets software practice: Lessons from the software development trenches. In *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE'07)*. IEEE, Los Alamitos, California, 265–268.

HEITMEYER, C., KIRBY, JR., J., LABAW, B., ARCHER, M., AND BHARADWAJ, R. 1998. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. Softw. Eng. 24,* 11, 927–948.

HENNESSY, M. AND MILNER, R. 1985. Algebraic laws for nondeterminism and concurrency. *J. ACM 32,* 1, 137–161.

HEYM, W. D. 1995. Computer program verification: Improvements for human reasoning. Ph.D. dissertation, Ohio State University, Columbus, OH.

HIERONS, R. M., BOGDANOV, K., BOWEN, J. P., CLAVELAND, R., DERRICK, J., DICK, J., GHERORGHE, M., HARMAN, M., KAPOOR, K., KRAUSE, P., LÜTTGEN, G., SIMMONS, A. J. H., VILKOMIR, S., WOODWARD, M. R., AND ZEDAN, H. 2009. Using formal specifications to support testing. *ACM Comput. Surv. 41,* 2, 9:1–9:76.

HOARE, C. A. R. 1969. An axiomatic basis for computer programming. *Commun. ACM 12,* 10, 576–580, 583.

HOARE, C. A. R. 1971. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, E. Engeler, Ed. Lecture Notes in Mathematics, vol, 18, Springer-Verlag, Berlin.

HOARE, C. A. R. 1972. Proof of correctness of data representations. *Acta Informatica 1,* 4, 271–281.

HOARE, C. A. R. 1985. *Communicating Sequential Processes*. Prentice-Hall, Inc., Englewood Cliffs, NJ.

HOARE, C. A. R. 2005. The verifying compiler, a grand challenge for computing research. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference*. Lecture Notes in Computer Science, vol. 3385. Springer-Verlag, Berlin, 78.

HOARE, C. A. R. AND HE, J. 1998. *Unifying Theories of Programming*. Prentice-Hall International, Englewood Cliffs, NJ.

HOARE, T. 2003. The verifying compiler: A grand challenge for computing research. *J. ACM 50,* 1, 63–69.

HOARE, T., LEAVENS, G. T., MISRA, J., AND SHANKAR, N. 2007. The verified software initiative: A manifesto. `http://qpq.csl.sri.com/vsr/manifesto.pdf`.

HOLZMANN, G. J. 1991. *Design and validation of computer protocols*. Prentice-Hall, Englewood Cliffs, NJ.

HOLZMANN, G. J. 1997. The model checker SPIN. *IEEE Trans. Softw. Eng. 23,* 5, 279–295.

HUISMAN, M. AND JACOBS, B. 2000. Java program verification via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering (FASE 2000)*, T. Maibaum, Ed. Lecture Notes in Computer Science, vol. 1783. Springer-Verlag, Berlin, 284–303.

JACKSON, D. 1995. Structuring Z specifications with views. *ACM Trans. Softw. Eng. Methodol. 4,* 4, 365–389.

JACKSON, D. 2006. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge, MA.

JACKY, J., VEANES, M., CAMPBELL, C., AND SCHULTE, W. 2008. *Model-Based Software Testing and Analysis with C#*. Cambridge University Press, Cambridge, U.K.

JACOBS, B., MÜLLER, P., AND PIESSENS, F. 2007. Sound reasoning about unchecked exceptions. In *Proceedings of the International Conference Software Engineering and Formal Methods (SEFM)*, M. Hinchey and T. Margaria, Eds. IEEE, Los Alamitos, CA, 113–122.

JACOBS, B. AND PIESSENS, F. 2008. The VeriFast program verifier. Tech. rep. CW-520, Department of Computer Science, Katholieke Universiteit Leuven.

JACOBS, B., PIESSENS, F., SMANS, J., LEINO, K. R. M., AND SCHULTE, W. 2008. A programming model for concurrent object-oriented programs. *ACM Trans. Program. Lang. Syst. 31,* 1, 1–48.

JONES, C. B. 1983. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst. 5,* 4, 596–619.

JONES, C. B. 1990. *Systematic Software Development Using VDM*, 2nd ed. International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ.

JONES, C. B. 1995. Partial functions and logics: A warning. *Inf. Process. Lett. 54,* 2, 65–67.

JONES, C. B. 2003. The early search for tractable ways of reasoning about programs. *IEEE Ann. History Comput. 25,* 2, 26–49.

JONES, C. B. 2006. Reasoning about partial functions in the formal development of programs. *Electron. Notes Theoret. Comput. Sci. 145*, 3–25.

KASSIOS, I. T. 2006. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Formal Methods (FM)*, E. S. J. Misra, T. Nipkow, Ed. Lecture Notes in Computer Science, vol. 4085. Springer-Verlag, Berlin, 268–283.

KHURSHID, S., PĂSĂREANU, C. S., AND VISSER, W. 2003. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, H. Garavel and J. Hatcliff, Eds. Lecture Notes in Computer Science, vol. 2619. 553–568.

KLIGERMAN, E. AND STOYENKO, A. 1992. Real-time Euclid: A language for reliable real-time systems. In *Real-Time Systems: Abstractions, Languages, and Design Methodologies*, K. M. Kavi, Ed. IEEE Computer Society Press, Los Alamitos, California, 455–463.

KOZEN, D. AND TIURYN, J. 1990. Logics of programs. In *Handbook of Theoretical Computer Science*, vol. *B: Formal Models and Semantics*. J. van Leewen, Ed. The MIT Press, New York, NY, 789–840.

KRAMER, J. AND MAGEE, J. 2006. *Concurrency: State Models & Java Programs, 2nd Edition*. Worldwide Series in Computer Science. John Wiley and Sons, Hoboken, NJ.

LAMPORT, L. 1989. A simple approach to specifying concurrent systems. *Commun. ACM 32,* 1, 32–45.

LAMPORT, L. 1994. The temporal logic of actions. *ACM Trans. Program. Lang. Syst. 16,* 3, 872–923.

LAMPSON, B. W., HORNING, J. L., LONDON, R. L., MITCHELL, J. G., AND POPEK, G. J. 1981. Report on the programming language Euclid. Tech. rep. CSL-81-12, Xerox Palo Alto Research Centers. Also *SIGPLAN Not.*, *12*, 2.

LARSEN, K. G., PETTERSSON, P., AND YI, W. 1997. UPPAAL in a Nutshell. *Int. J. Softw. Tools Technol. Transfer 1,* 1-2, 134–152.

LEAVENS, G. T. 2006. JML's rich, inherited specifications for behavioral subtypes. In *Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM)*, Z. Liu and H. Jifeng, Eds. Lecture Notes in Computer Science, vol. 4260. Springer-Verlag, New York, NY, 2–34.

LEAVENS, G. T., BAKER, A. L., AND RUBY, C. 2006. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Softw. Eng. Notes 31,* 3, 1–38.

LEAVENS, G. T., CHEON, Y., CLIFTON, C., RUBY, C., AND COK, D. R. 2005. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program. 55,* 1-3, 185–208.

LEAVENS, G. T. AND MÜLLER, P. 2007. Information hiding and visibility in interface specifications. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, Los Alamitos, CA, 385–395.

LEAVENS, G. T. AND NAUMANN, D. A. 2006. Behavioral subtyping, specification inheritance, and modular reasoning. Tech. rep. 06-20b, Department of Computer Science, Iowa State University, Ames, IA.

LEAVENS, G. T., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D. R., MÜLLER, P., KINIRY, J., CHALIN, P., AND ZIMMERMAN, D. M. 2009. JML Reference Manual. `http://www.jmlspecs.org`.

LEAVENS, G. T. AND WEIHL, W. E. 1995. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica 32,* 8, 705–778.

LEAVENS, G. T. AND WING, J. M. 1997. Protective interface specifications. In *Proceedings of the 7th International Joint Conference on Theory and Practice of Software Development (TAPSOFT'97)*, M. Bidoit and M. Dauchet, Eds. Lecture Notes in Computer Science, vol. 1214. Springer-Verlag, New York, NY, 520–534.

LEINO, K. R. M. 1995. Toward reliable modular programs. Ph.D., dissertation California Institute of Technology. Tech. rep. Caltech-CS-TR-95-03.

LEINO, K. R. M. 1998. Data groups: Specifying the modification of extended state. In *Proceedings of the Annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). SIGPLAN Not.*, *33*, 10. ACM, New York, NY, 144–153.

LEINO, K. R. M. 2010. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning, 16th International Conference*. Lecture Notes in Computer Science. Springer-Verlag. Forthcoming.

LEINO, K. R. M. AND MÜLLER, P. 2004. Object invariants in dynamic contexts. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, M. Odersky, Ed. Lecture Notes in Computer Science, vol. 3086. Springer-Verlag, Berlin, 491–516.

LEINO, K. R. M. AND MÜLLER, P. 2006. A verification methodology for model fields. In *Proceedings of the European Symposium on Programming (ESOP)*, P. Sestoft, Ed. Lecture Notes in Computer Science, vol. 3924. Springer-Verlag, New York, NY, 115–130.

LEINO, K. R. M. AND MÜLLER, P. 2008. Verification of equivalent-results methods. In *Proceedings of the 17th European Symposium on Programming, Languages and Systems (ESOP)*, S. Drossopoulou, Ed. Lecture Notes in Computer Science, vol. 4960. Springer-Verlag, Berlin, 307–321.

LEINO, K. R. M. AND MÜLLER, P. 2009. A basis for verifying multi-threaded programs. In *Proceedings of the 18th European Symposium on Programming, Languages and Systems (ESOP)*, G. Castagna, Ed. Lecture Notes in Computer Science, vol. 5502. Springer-Verlag, Berlin, 378–393.

LEINO, K. R. M., MÜLLER, P., AND SMANS, J. 2009. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*, A. Aldini, G. Barthe, and R. Gorrieri, Eds. Lecture Notes in Computer Science, vol. 5705. Springer-Verlag, Berlin, 195–222.

LEINO, K. R. M., MÜLLER, P., AND WALLENBURG, A. 2008. Flexible immutability with frozen objects. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*, N. Shankar and J. Woodcock, Eds. Lecture Notes in Computer Science, vol. 5295. Springer-Verlag, Berlin, 192–208.

LEINO, K. R. M. AND NELSON, G. 2002. Data abstraction and information hiding. *ACM Trans. Program. Lang. Syst. 24,* 5, 491–553.

LEINO, K. R. M., POETZSCH-HEFFTER, A., AND ZHOU, Y. 2002. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02). SIGPLAN Not. 37*, 5. ACM, New York, NY, 246–257.

LEINO, K. R. M. AND RÜMMER, P. 2010. A polymorphic intermediate verification language: Design and logical encoding. In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference*, J. Esparza and R. Majumdar, Eds. Lecture Notes in Computer Science, vol. 6015. Springer-Verlag, 312–327.

LEINO, K. R. M., SAXE, J. B., AND STATA, R. 1999. Checking Java programs via guarded commands. In *Formal Techniques for Java Programs (FTfJP)*, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, Eds. Tech. rep. 251. FernUniversität Hagen, Hagen, Germany. Also available as Tech. note 1999-002, Compaq Systems Research Center.

LEINO, K. R. M. AND SCHULTE, W. 2004. Exception safety for C#. In *Proceedings of the 2nd International Conference on Software Engineering and Formal Methods*, J. R. Cuellar and Z. Liu, Eds. IEEE, Los Alamitos, CA, 218–227.

LEINO, K. R. M. AND SCHULTE, W. 2007. Using history invariants to verify observers. In *Programming Languages and Systems (ESOP)*, R. D. Nicola, Ed. Lecture Notes in Computer Science, vol. 4421. Springer-Verlag, Berlin, 80–94.

LEINO, K. R. M. AND WALLENBURG, A. 2008. Class-local object invariants. In *Proceedings of the 1st India Software Engineering Conference (ISEC'08)*. ACM, New York, NY.

LEROY, X. 2006. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, J. G. Morrisett and S. L. P. Jones, Eds. ACM, New York, NY, 42–54.

LEVESON, N. G., HEIMDAHL, M. P. E., AND REESE, J. D. 1999. Designing specification languages for process control systems: Lessons learned and steps to the future. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Imposium on Foundations of Software Engineering*, O. Nierstrasz and M. Lemoine, Eds. Lecture Notes in Computer Science, vol. 1687. Springer-Verlag, Berlin, 127–145. Also *ACM SIGSOFT Softw. Eng. Notes 24*, 6.

LHOTKA, R. 2008. *Expert C# 2008 Business Objects*. Apress, New York, NY.

LISKOV, B. 1988. Data abstraction and hierarchy. *SIGPLAN Not. 23,* 5. 17–34. Revised version of the keynote address given at the 2nd Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications.

LISKOV, B. AND GUTTAG, J. 1986. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, MA.

LISKOV, B. H. AND WING, J. M. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16,* 6, 1811–1841.

LOECKX, J. AND SIEBER, K. 1987. *The Foundations of Program Verification, 2nd Edition*. John Wiley and Sons, New York, NY.

LU, Y., POTTER, J., AND XUE, J. 2007. Validity invariants and effects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, E. Ernst, Ed. Lecture Notes in Computer Science, vol. 4609. Springer-Verlag, Berlin, 202–226.

LUCKHAM, D. 1990. *Programming with Specifications: An Introduction to Anna, A Language for Specifying Ada Programs*. Texts and Monographs in Computer Science. Springer-Verlag, New York, NY.

LUCKHAM, D. AND VON HENKE, F. W. 1985. An overview of Anna—A specification language for Ada. *IEEE Softw. 2,* 2, 9–23.

MAGEE, J. AND KRAMER, J. 2005. Model-based design of concurrent programs. In *Communicating Sequential Processes: The First 25 Years*. Lecture Notes in Computer Science, vol. 3525. Springer-Verlag, Berlin, 211–219.

MANASSE, M. S. AND NELSON, C. G. 1984. Correct compilation of control structures. Tech. rep., AT&T Bell Laboratories.

MANNA, Z. AND PNUELI, A. 1992. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, NY.

MARCEAU, C. 1994. Penelope reference manual, version 3-3. Tech. rep. TM-94-0040, Odyssey Research Associates, Inc., Ithaca, NY.

MARCHÉ, C. 2009. The Krakatoa tool for deductive verification of Java programs. Winter School on Object-Oriented Verification, Viinistu, Estonia. `http://krakatoa.lri.fr/ws/`.

MARCHÉ, C. AND PAULIN-MOHRING, C. 2005. Reasoning about Java programs with aliasing and frame conditions. In *Proceedings of the 18th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs'05)*, J. Hurd and T. F. Melham, Eds. Lecture Notes in Computer Science, vol. 3603. Springer-Verlag, Berlin, 179–194.

MARCHÉ, C., PAULIN-MOHRING, C., AND URBAIN, X. 2004. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *J. Logic Algebraic Program. 58,* 1-2, 89–106.

MARTIN-LÖF, P. 1985. Constructive mathematics and computer programming. In *Mathematical Logic and Programming Languages*. Prentice-Hall, Inc., Englewood Cliffs, NJ., 167–184.

MCCONNELL, S. 1993. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, Buffalo, NY.

MEYER, A. R. AND SIEBER, K. 1988. Towards fully abstract semantics for local variables: Preliminary report. In *Proceedings of the* 15th *Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 191–203.

MEYER, B. 1992. Applying "design by contract". *Comput. 25,* 10, 40–51.

MEYER, B. 1997. *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, New York, NY.

MILNE, G. AND MILNER, R. 1979. Concurrent processes and their syntax. *J. ACM 26,* 2, 302–321.

MILNER, R. 1990. Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science, vol. B: Formal Models and Semantics.* J. van Leewen, Ed. The MIT Press, New York, NY, 1201–1242.

MILNER, R., PARROW, J., AND WALKER, D. 1992. A calculus of mobile processes, parts I and II. *Inform. Comput. 100*, 1–77.

MITRA, S. 1994. Object-oriented specification in VDM++. In *Object-Oriented Specification Case Studies*, K. Lano and H. Haughton, Eds. The Object-Oriented Series. Prentice-Hall, New York, NY, 130–136.

MORGAN, C. 1990. *Programming from Specifications*. Prentice Hall International, Hempstead, U.K.

MORGAN, C. 1994. *Programming from Specifications, 2nd Edition*. Prentice Hall International, Hempstead, U.K.

MORGAN, C. AND VICKERS, T., Eds. 1994. *On the Refinement Calculus*. Formal Approaches of Computing and Information Technology Series. Springer-Verlag, New York, NY.

MORRIS, J. B. 1980. Programming by successive refinement of data abstractions. *Softw. Pract. Exp. 10,* 4, 249–263.

MOSSAKOWSKI, T., HAXTHAUSEN, A. E., SANELLA, D., AND TARLECKI, A. 2008. CASL—The Common Algebraic Specification Language. In *Logics of Specification Languages,* Springer-Verlag, Berlin, 241–298.

MÜLLER, P. 2002. *Modular Specification and Verification of Object-Oriented Programs*. Lecture Notes in Computer Science, vol. 2262. Springer-Verlag, Berlin.

MÜLLER, P., POETZSCH-HEFFTER, A., AND LEAVENS, G. T. 2003. Modular specification of frame properties in JML. *Concurrency Comput. Pract. Exp. 15,* 2, 117–154.

MÜLLER, P., POETZSCH-HEFFTER, A., AND LEAVENS, G. T. 2006. Modular invariants for layered object structures. *Sci. Comput. Program. 62,* 3, 253–286.

NAUMANN, D. A. 2001. Calculating sharp adaptation rules. *Inf. Process. Lett. 77*, 201–208.

NAUMANN, D. A. 2007. Observational purity and encapsulation. *Theor. Comput. Sci. 376,* 3, 205–224.

NIELSON. 1996. Annotated type and effect systems. *ACM Comput. Surv. 28,* 2, 344–345.

OBJECT MANAGEMENT GROUP. 1992. *The Common Object Request Broker: Architecture and Specification*, 1.1 ed. Object Management Group, Inc., Framingham, MA.

OGDEN, W. F., SITARAMAN, M., WEIDE, B. W., AND ZWEBEN, S. H. 1994. Part I: The RESOLVE framework and discipline—A research synopsis. *ACM SIGSOFT Softw. Eng. Notes 19,* 4, 23–28.

O'HEARN, P., REYNOLDS, J., AND YANG, H. 2001. Local reasoning about programs that alter data structures. In *Proceedings of the Workshop on Computer Science Logic*. Lecture Notes in Computer Science, vol. 2142. Springer-Verlag, Berlin, 1–19.

O'HEARN, P. W. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci. 375,* 1-3, 271–307.

O'HEARN, P. W., YANG, H., AND REYNOLDS, J. C. 2004. Separation and information hiding. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, N. D. Jones and X. Leroy, Eds. ACM, New York, NY, 268–280.

OLDEROG, E. 1983. On the notion of expressiveness and the rule of adaptation. *Theor. Comput. Sci. 24*, 337–347.

OLDEROG, E.-R. 2008. Automatic verification of combined specifications: An overview. *Electron. Notes Theor. Comput. Sci. 207*, 3–16.

OMG. 2006. Object constraint language specification, version 2.0. `http://tinyurl.com/k7rfm`.

OUAKNINE, J. AND WORRELL, J. 2005. On the decidability of metric temporal logic. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*. IEEE Computer Society, Los Alamitos, CA, 188–197.

OWICKI, S. S. 1975. *Axiomatic Proof Techniques for Parallel Programs*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, NY.

PARKINSON, M. AND BIERMAN, G. 2005. Separation logic and abstraction. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, J. Palsberg and M. Abadi, Eds. ACM, New York, NY, 247–258.

PARKINSON, M. AND BIERMAN, G. 2008. Separation logic, abstraction and inheritance. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, P. Wadler, Ed. ACM, New York, NY, 75–86.

PARKINSON, M., BORNAT, R., AND CALCAGNO, C. 2006. Variables as resource in Hoare logics. In *Proceedings of the 21th IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, Los Alamitos, CA, 137–146.

PARKINSON, M. J. 2005. Local reasoning for Java. Ph.D. dissertation. Tech. rep. 654, University of Cambridge Computer Laboratory. Cambridge, U.K.

PARNAS, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15,* 12, 1053–1058.

PARNAS, D. L. 1993. Predicate logic for software engineering. *IEEE Trans. Softw. Eng. 19,* 9, 856–862.

PARTSCH, H. AND STEINBRÜGGEN, R. 1983. Program transformation systems. *Comput. Sur. 15,* 3, 199–236.

PERL, S. E. AND WEIHL, W. E. 1993. Performance assertion checking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. ACM, New York, NY, 134–145.

PETERSON, J. L. 1977. Petri nets. *ACM Comput. Surv. 9,* 3, 221–252.

PETERSON, J. L. 1981. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, NJ.

PILAUD, D., HALBWACHS, N., AND PLAICE, J. A. 1987. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL'87)*. ACM, New York, NY, 178–188.

PILKIEWICZ, A. AND POTTIER, F. 2009. The essence of monotonic state. `http://gallium.inria.fr/~fpottier/publis/pilkiewicz-pottier-monotonicity-2009.pdf`.

POETZSCH-HEFFTER, A. 1997. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, Munich, Germany.

POETZSCH-HEFFTER, A. AND MÜLLER, P. 1999. A programming logic for sequential Java. In *Proceedings of the European Symposium on Programming (ESOP'99)*, S. D. Swierstra, Ed. Lecture Notes in Computer Science, vol. 1576. Springer-Verlag, Berlin, 162–176.

REYNOLDS, J. 2000. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science, Proceedings of the Oxford-Microsoft Symposium in Honor of Sir Tony Hoare*.

REYNOLDS, J. C. 1981. *The Craft of Programming*. Prentice-Hall International, Englewood Cliffs, NJ.

REYNOLDS, J. C. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, CA.

RODRÍGUEZ, E., DWYER, M. B., FLANAGAN, C., HATCLIFF, J., LEAVENS, G. T., AND ROBBY. 2005. Extending JML for modular specification and verification of multi-threaded programs. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, A. P. Black, Ed. Lecture Notes in Computer Science, vol. 3586. Springer-Verlag, Berlin, 551–576.

ROSCOE, A. W. 1994. Model-checking CSP. In *A Classical Mind: Essays in Honour of C. A. R. Hoare*. Prentice Hall International Ltd., Hertfordshire, U.K., 353–378.

RUBY, C. AND LEAVENS, G. T. 2000. Safely creating correct subclasses without seeing superclass code. In *ACM Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*. *SIGPLAN Not. 35*, 10. ACM, New York, NY, 208–228.

RUDICH, A., DARVAS, Á., AND MÜLLER, P. 2008. Checking well-formedness of pure-method specifications. In *Formal Methods*, J. Cuéllar, T. S. E. Maibaum, and K. Sere, Eds. Lecture Notes in Computer Science, vol. 5014. Springer-Verlag, Berlin, 68–83.

RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. 1999. *The Unified Modeling Language Refrence Manual*. Object Technology Series. Addison Wesley Longman, Reading, MA.

SALCIANU, A. AND RINARD, M. 2005. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation*, R. Cousot, Ed. Lecture Notes in Computer Science, vol. 3385. Springer-Verlag, Berlin, 199–215.

SCHMIDT, D. A. 1994. *The Structure of Typed Programming Languages*. Foundations of Computing Series. The MIT Press, Cambridge, MA.

SEN, K., MARINOV, D., AND AGHA, G. 2005. CUTE: A concolic unit testing engine for C. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering*, 263–272.

SHANER, S. M., LEAVENS, G. T., AND NAUMANN, D. A. 2007. Modular verification of higher-order methods with mandatory calls specified by model programs. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, New York, NY, 351–367.

SMANS, J., JACOBS, B., AND PIESSENS, F. 2008a. Implicit dynamic frames. In *Formal Techniques for Java-like Programs*, M. Huisman, Ed. Radboud University, Nijmegen, 1–12. Tech. rep. ICIS-R08013, Radboud University, Nijmegen.

SMANS, J., JACOBS, B., AND PIESSENS, F. 2008b. VeriCool: An automatic verifier for a concurrent object-oriented language. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, G. Barthe and F. S. de Boer, Eds. Lecture Notes in Computer Science, vol. 5051. Springer-Verlag, Berlin, 220–239.

SMANS, J., JACOBS, B., AND PIESSENS, F. 2009. Implicit dynamic frames: Combining dynamic frames and separation logic. In *Object-Oriented Programming (ECOOP)*, S. Drossopoulou, Ed. Lecture Notes in Computer Science, vol. 5653. Springer-Verlag, Berlin, 148–172.

SMANS, J., JACOBS, B., PIESSENS, F., AND SCHULTE, W. 2008. An automatic verifier for Java-like programs based on dynamic frames. In *Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science, vol. 4961. Springer-Verlag, Berlin, 261–275.

SPIVEY, J. 1989. An introduction to Z and formal specifications. *Softw. Eng. J. 4,* 1, 40–50.

STOYENKO, A. 1992. The evolution and state-of-the-art of real-time languages. In *Real-Time Systems: Abstractions, Languages, and Design Methodologies*, K. M. Kavi, Ed. IEEE Computer Society Press, Los Alamitos, California, 394–416.

VAN LAMSWEERDE, A. 2000. Requirements engineering in the year 00: A research perspective. In *Proceedings of the 22nd International Conference on Software Engineering*. ACM Press, New York, NY, 5–19.

VON OHEIMB, D. 2001. Analyzing Java in Isabelle/HOL: Formalization, type safety and Hoare logic. Ph.D. disssetation, Technische Universität München.

WALRATH, K., CAMPIONE, M., HUML, A., AND ZAKHOUR, S. 2004. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Prentice Hall, Upper Saddle River, NJ.

WARMER, J. AND KLEPPE, A. 1999. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley Longman, Reading, MA.

WING, J. M. 1987. Writing Larch interface language specifications. *ACM Trans. Program. Lang. Syst. 9,* 1, 1–24.

WING, J. M. 1990. A specifier's introduction to formal methods. *Comput. 23,* 9, 8–24.

WINSKEL, G. 1993. *The Formal Semantics of Programming Languages*. Foundations of Computer Science Series. The MIT Press, Cambridge, MA.

WIRSING, M. 1990. Algebraic specification. In *Handbook of Theoretical Computer Science*, *vol. B: Formal Models and Semantics*, J. van Leewen, Ed. The MIT Press, New York, NY, 675–788.

WOODCOCK, J., SAALTINK, M., AND FREITAS, L. 2009. Unifying theories of undefinedness. In *Summer School Marktoberdorf 2008: Engineering Methods and Tools for Software Safety and Security*. NATO ASI Series F. IOS Press, Amsterdam. Forthcoming.

XU, Q., DE ROEVER, W. P., AND HE, J. 1997. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects Comput. 9,* 2, 149–174.

ZEE, K., KUNCAK, V., AND RINARD, M. C. 2008. Full functional verification of linked data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, R. Gupta and S. P. Amarasinghe, Eds. ACM, New York, NY, 349–361.

Received October 2010; accepted January 2011