Behavioral Interface Specification Languages

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

In this paper we survey detailed design specification languages.

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: Assertion, Bandera, Eiffel, interface specification language, guarantee condition, invariant, JML, model checking, postcondition, precondition, rely condition, separation logic, Spec#, SPARK, temporal logic, VDM,

1. INTRODUCTION

Specifications that document the interfaces of program parts [Guttag et al. 1993; Jones 1990; Meyer 1997; Wing 1987; 1990] can play a central role in programming,

The work of 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 Leavens was supported in part by the US National Science Foundation under grant CNS 08-08913. The work of Parkinson was supported in part by a RAEng/EPSRC research fellowship and EPSRC grant EP/F019394.
DRAFT of `$Id: testluximono.tex 99 2009-01-04 01:05:49Z leino $`

recording the intended behavior of program parts. This information is key for maintenance, since code alone cannot reveal what was intended. When debugging one can use specifications to isolate faults and assign blame [Findler and Felleisen 2002; Liskov and Guttag 1986; Meyer 1992]. Specifications can also define a standard of correctness for testing.

Formal specifications have additional benefits. Formal, *i.e.*, mathematically precise, notation makes them unambiguous, less dependent on cultural norms[1], and thus less likely to be misunderstood. Formal specifications can help automate testing, both in helping decide on test results [Cheon and Leavens 2002; Peters and Parnas 1998] and in creation of test data [Bernot et al. 1991; Chang et al. 1996; Crowley et al. 1996; Jalote 1992; Korel and Al-Yami 1998; Richardson 1994; Sankar and Hayes 1994], and are a key ingredient in modular verification. During development, formal specifications can also serve as a starting place for transformational development [Abadi and Lamport 1988; Abrial 1996; Hehner 1993; Jones 1990; Morgan 1994; Morgan and Vickers 1994; Morris 1980; Partsch and Steinbrüggen 1983] or the search for reusable components [Zaremski and Wing 1997].

In this survey, we focus mainly on 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 known programming faults to the satisfaction of all properties that the program must satisfy. We discuss many examples of such properties below.

We intend this survey to be interesting for readers who want to:

—learn ways to specify or describe software properties, *e.g.*, for use by verification or static analysis tools, or

—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]. This goal, and space limitations, motivate our focus on detailed design specifications for code.

1.1 Brief Background

The term *specification* generally means a precise description of the behavior of some artifact, such as an abstract data type. Specifications are often used to record agreements, or contracts [Liskov and Guttag 1986; Meyer 1992], between a software artifact's implementors and its clients. We use *formal specification* for specifications with a mathematically precise semantics. *Verification* means proving that an implementation satisfies a particular specification in every possible execution. Usually such a proof is accomplished by static reasoning, but it is possible to combine static reasoning with dynamic checks (such as runtime assertion checking) [Geller 1978; Flanagan 2006].

Specifications are used in many different parts of different software processes. Requirements on a system's software as a whole (the "machine" [Jackson 1995]) are recorded in a requirements specification [van Lamsweerde 2000; Heitmeyer et al. 2007]. Such requirements specifications are crucial for software engineering, since nothing else matters if the software is solving the wrong problem [Brooks 1987].

[1]For example, a "billion," means 10^9 in the US, but 10^{12} in Britan [Liskov and Guttag 1986].

The architecture of a program can also be recorded in an architectural specification, documenting permitted connections between components [Aldrich et al. 2002; Garlan et al. 2000]. Domain analysis and modeling produces precise mathematical descriptions of data and operations that form a useful vocabulary for writing specifications and exploring design decisions in a particular application area [Hayes 1993; Jackson and Rinard 2000; Jackson 2006; Spivey 1989]. Algebraic specification techniques are particularly well suited to describing such domain models [Bidoit et al. 1991; CoFI (The Common Framework Initiative) 2004; Goguen et al. 1978; Guttag and Horning 1978; Guttag et al. 1993; Wand 1979; Wirsing 1990] (although they have other uses as well [Boehm 1985; Goguen and Malcolm 1996; Hoare et al. 1987]).

Our focus in this paper is on a lower level of abstraction: recording detailed design decisions about program modules that will be useful in coding. Wing [Wing 1987; 1990] and Lamport [Lamport 1989] called such specifications *interface specifications*, since they document both the interface between 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 some authors use the term *behavioral interface specification* [Cheon and Leavens 1994a] to emphasize the behavioral component of such specifications, such as pre- and postconditions.

Specifications can be stated in a variety of forms, some of which more tightly constrain software than others. An ideal might be a specification that describes all the required *functional behavior* (*i.e.*, inputs and outputs) of an entire software system. Ideally, such a specification would be a refinement of a requirements specification. A specification B *refines* specification A if every implementation that satisfies B also satisfies A. When one has the luxury of designing from such a functional specification of a system, careful processes can yield higher productivity and leave behind a rational design history [Barnes et al. 2006; Chapman 2000; Hall and Chapman 2002; King et al. 1999]. However, one is often faced with an extant program and nothing like a complete specification of a system's functional behavior; in such cases one might wish to verify that the program satisfies a weaker, implicit specification — that certain programming errors (*e.g.*, dereferencing a null pointer) cannot occur [Sites 1974; German 1978; Detlefs et al. 1998]. A basic distinction is between such *safety properties*, which say that nothing bad happens, and liveness properties [Manna and Pnueli 1992]. A *liveness* property says that something good eventually happens, for example, that a system eventually responds to a request. Another kind of specification concerns resource usage, such as timing constraints or constraints on maximum memory usage.

A *specification language*, which is a formal language capable of recording properties, can be designed to be general or specialized. General specification languages, such as Z [Hayes 1993; Spivey 1989] and OCL [OMG 2006; Warmer and Kleppe 1999], are not tailored to any specific methodology or verification tool. However, specification languages are often specialized. Some are specialized to support a particular methodology, such as Eiffel's support for design by contract (and dynamic checking) [Meyer 1997] or RESOLVE's [Edwards et al. 1994] avoidance of

aliasing by using swapping instead of assignment [Harms and Weide 1991]. Many others are specialized to support some particular automated verification technique such as Promela's support for the SPIN model checker [Holzmann 1991]. However, with the recognition that there are interesting synergies between several verification techniques, such as static and dynamic checking [Ernst 2003; Flanagan 2006], some languages, such as JML [Burdy et al. 2005; Leavens et al. 2008], have been designed to support multiple tools.

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.

Verification technology is also closely tied to semantics. A *verification logic* is a formal reasoning system that allows proofs 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 (such as denotational [Schmidt 1986; 1994; Scott and Strachey 1971] or structural operational [Astesiano 1991; Hennessy 1990; Plotkin 1977; 1981] 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 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; Edmund M. Clarke et al. 1999; Holzmann 1997] uses state space exploration techniques to verify programs, which is directly based on the semantics of finite state machines.

1.2 Kinds of Specification Languages

We organize the survey around classes of properties. As a prelude to discussion of the classes of properties we survey, we present a broader categorization of possible kinds of properties and their relation to known specification languages.

—Requirements-level properties (whole system specifications), and requirements specification languages such as SCR [Heitmeyer et al. 1998; Heitmeyer and Jeffords 2007; Jeffords and Heitmeyer 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], parts of the UML [Arlow and Neustadt 2005; Rumbaugh et al. 1999] such as Statecharts [Harel et al. 1990; Harel 1987], etc.

—Analysis-level properties that express concepts in a domain. These include:
  —*Functional behavior* properties describe the (data) values associated with system operations or state changes and that include types that may take on a potentially infinite number of values. Such properties typically describe the relationships between inputs and outputs, and thus typically do not involve more than two states. Analysis specification languages that describe func-

tional 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.

—*Temporal* properties describe a finite set of events (*i.e.*, states) and properties of a system's (potentially infinite) sequences of events. Such properties typically do not involve infinite sets of data values. 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; 1991; Milner et al. 1992].

—*Resource* properties state 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 [Alur and Henzinger 1992; Jahanian and Mok 1986]. Such 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; Halbwachs 2005; Pilaud et al. 1987], and the duration calculus [Hansen 2008]. [Bellini et al. 2000] survey specification languages build on temporal logic for real-time system specification.

—Blends of the above (values and events), with languages such as CSP [Brookes et al. 1984; Hoare 1985; Roscoe 1994], RAISE [Nielsen et al. 1989; George and Haxthausen 2008].

—Properties of code and interface specification languages. These include:

—Functional properties of data, such as pre- and postconditions, invariants, frame axioms, etc. This includes properties specified 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; Lampson and Burstall 1988; Constable et al. 1986; Schmidt 1994; Martin-Löf 1985]. Functional properties of data for imperative languages often hold only at certain program points, so design-by-contract style specification languages 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; 1997] 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] (such as LM3

[Jones 1991], LCL [Guttag and Horning 1991; Tan 1995], Larch/C++ [Cheon and Leavens 1994b], and Larch/Smalltalk [Cheon and Leavens 1994a]), 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; Leavens et al. 2005]. The Object Constraint Language of the UML [Warmer and Kleppe 1999; OMG 2006] also fits in this style. This style also includes more property-oriented specification languages, such as B [Abrial 1996; Lano 1996] and Event-B [Cansell and Méry 2008]. There is also work that blends these two styles, often doing runtime checking when type-specified properties cannot be statically proven by a type checker [Flanagan 2006; Cartwright and Felleisen 1996].

—Temporal properties of events in code, such as those expressible in various forms of temporal logic. 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].

JH comment: I think there are additional languages that we can insert here such as those used for specifying temporal properties in run-time monitoring, type-state notations, etc.

GTL comment: Great, please do so.

—Timing properties for events in code, including the PSpec language [Perl and Weihl 1993] and Real-time Euclid [Kligerman and Stoyenko 1992; Stoyenko 1992].

In certain domains one can also declaratively specify computations or computational artifacts, using specification constructs like those described above or using more specialized notations. For example, one can specify the semantics of a programming language using the conventions of denotational or operational semantics [Schmidt 1994; Winskel 1993]. Another example is that verification tools can make use of specialized languages for stating proof obligations and a program's semantics, such as Boogie [Leino 2008] and Why [Filliâtre 2003].

1.3 Scope

We focus on *specification languages* (or "assertion languages") that can be used to document the behavior of executable code modules and that can be mechanized to aid the Verified Software Initiative. This includes languages that are able to specify behavior across a range of levels in the software development process as long as that range includes executable code. Most such languages are usually not designed for the specification of the behavior of an entire application or an application's environment, but rather are designed to specify the behavior of individual modules and the environmental assumptions within a program that such modules make. In particular we consider interface specification languages that are able to document both the behavior of code modules and their interfaces to other program modules [Lamport 1989; Wing 1987]. Thus in particular, we focus on specification languages that are able to document detailed designs at the same level of abstraction as a programming language.

However, we do not limit ourselves to interface specification languages or to languages that are *only* capable of documenting detailed designs. That is, we include languages that can be used in stepwise refinement efforts, perhaps even starting with user-level requirements. However, as we will discuss below, we exclude languages that are not able to document the behavior of code.

Due to space restrictions, we focus on assertion languages for sequential programs that describe their functional behavior.

1.4 Outline

In the remainder of this survey, we have one section for each kind of property.

2. FUNCTIONAL (INPUT-OUTPUT) BEHAVIOR

The most basic notions of behavioral specifications capture a program's *functional behavior*. Functional specifications may describe a relationship between a program's (or subprogram's) inputs and outputs. For example, a functional 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. Functional 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 non-empty before the sorting process begins. Another common form of constraint is an *invariant* — a predicate that must be hold in every program state encountered during execution. For example, in a sorting algorithm, one might have an invariant stating that an index variable for the array being sorted must always lie within the bounds of the array.

This section overviews basic forms of code-level functional specifications along with notions of program architecture, encapsulation, and abstraction that are directly related to such 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 now sketch a simple semantic model of programs that we will use to ground the discussion of specification semantics below. A *program store* σ for $\mathcal{P}$ is a mapping from $\mathcal{P}$'s variables to values. Intuitively, a store represents the programmer's view of the computer's memory as it stores the current value associated with each program variable. As $\mathcal{P}$ executes, a *program counter* pc indicates the current command be to executed. A *program state* is a tuple (pc, σ) consisting of the current value of the program counter pc and the current store σ.

A *store predicates* is a boolean formula over variable values, *e.g.*, $X > 1 \vee Y = 5$. We say that a store predicate P *holds* for the values of variables in a particular store σ (denoted $\sigma \models P$) when the values of variables as given by σ satisfy the constraints given by P. A store predicate P can also be viewed as characterizing a

set of stores $[\![P]\!]$ – namely, the set of stores for which P holds.

2.1 In-line 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 non-negative values. The developer can use an assertion to specify this assumption/belief about the functional 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 SPARK Ada assertion above) and are recognized and processed by verification tools that understand the special comment syntax (`--#` indicates a SPARK specification). In other cases (such as with Java assertions), assertions are written as executable code that be may executed during testing and removed by a pre-processor once testing is completed.

Although assertions can be checked by static analysis tools, they are typically 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 a system-generated error message is displayed on the system console. In additional to this operational benefit of assertions, assertion information can aid subsequent code maintenance (perhaps carried out by someone other than then original developer) by expressing assumptions and important aspects of the intended functionality of the code.

2.1.2 *Semantic Foundations.* Formally, an assertion a is a store predicate P_a that must hold on the program store that is current when execution reaches the point in the program where the assertion is written. When one considers the program counter position at which the assertion is written as part of its definition, then an assertion $a = (pc_a, P_a)$ is an invariant that must hold in each state (pc, σ); where (pc_a, P_a) holds in state (pc, σ) if $pc = pc_a$ implies $\sigma \models P_a$.

2.1.3 *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 (see, *e.g.*, [McConnell 1993]).

Clarke and Rosenblum [Clarke and Rosenblum 2006] give a detailed historical perspective on the development and use of runtime assertion checking. (See also Jones's article on the early history of reasoning about programs [Jones 2003].) 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, in-line assertions can be included in almost any language by defining an **assert** construct as a procedure or macro.

A key issue for tools is when and how to check assertions.

—A number of program checkers and verifiers analyze in-line assertions statically. For example, ESC/Java attempts to prove that each in-line assertion will hold in all executions.

—If both static and dynamic checking are supported by tools, then it is useful to distinguish between assertions that one expects a static program checker to verify and assertions that cannot be verified in the given context but that are needed for the rest of the verification. JML and Spec# offer a way to distinguish these, using an **assert** statement for properties to be verified and an `assume` statement for properties that are declared to be true. When assert statements have been statically verified, one may want to disable run-time checking for them while always enabling the run-time checks of assume statements.

—Run-time checking of in-line assertions is effective in finding unexpected behavior during the development and testing of a program.

GTL comment: Citation for that?

KRML comment: I think *Code Complete* mentions this, but I'll need to double check at the office.

However, when the quality of a program is high enough to deploy the program, developers often want to trade the additional checking for improved performance, which can be achieved by disabling run-time assertion checking.

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 assertions. A precondition is an assertion stated at the beginning a procedure that specifies conditions that must hold true whenever the procedure is called. A postcondition is an assertion stated at the end of procedure that specifies conditions that must hold immediately after the procedure completes its execution.

The SPARK function `Find_Index_Pos` of Figure 1 illustrates the use of pre/postconditions in a simple function that 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 code 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 at the header of the function.

The precondition states that the value of `X` must be in the array when the function 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 be richer than language of program expressions (program expressions may not contain existential quantification).

```
function Find_Index_Pos (X : 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;

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. 1. SPARK illustrations of structured assertions

In the postcondition, the construct **return** Z names the return value Z and the 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 sub-program can be viewed as summarizing the sub-program's functional behavior in the sense that the associated assertions describe properties of states flowing into and out of the sub-program. Summaries can vary in their precision. For example, dropping the constraint (2) in the postcondition above still yields a valid summary of the associated function implementation, but it is a less precise summary because it doesn't capture the fact that the index value returned corresponds to the first occurrence of X.

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

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 store predicates. The triple is *valid* iff for any store σ that satisfies P, executing C on σ yields a store σ' that satisfies Q. Since $[\![P]\!]$ and $[\![Q]\!]$ can be viewed as characterizing sets of stores, a triple can be viewed as summarizing the input/output behavior of C in terms of the set of output stores $[\![Q]\!]$ that result from input stores in $[\![P]\!]$.

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 stores – a fact that is perhaps more easily grasped when considering that $P \Rightarrow Q$ holds when $[\![P]\!] \subseteq [\![Q]\!]$. Store predicates can be viewed as abstractions that summarize program state information, and they can be arranged in a natural approximation lattice based on the entailment relation as an ordering.

This approximation ordering on store predicates can be used to define an approximation ordering on pre/postconditions pairs (sub-program contracts) as illustrated in the following diagram.

$$\begin{array}{lccccl} \textit{less precise} & & \{P\} & C & \{Q\} & \\ & \textit{refine} \downarrow & \Downarrow & & \Uparrow & \textit{abstract} \uparrow \\ \textit{more precise} & & \{P'\} & C' & \{Q'\} & \end{array}$$

Let κ represent the top contract $\{P\} \cdot \{Q\}$ and κ' represent the bottom contract $\{P'\} \cdot \{Q'\}$. κ' is said to be a *refinement* of κ (alternatively, κ is an abstraction of κ') if $Q' \Rightarrow Q$ and $P \Rightarrow P'$ [Back 1978].

From the point of view of clients that might call sub-programs with contracts κ' or κ, if κ' refines κ then a sub-program C' that satisfies κ' can be used in any calling context where C satisfying κ 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 $P \Rightarrow P'$ ($[\![P]\!] \subseteq [\![P']\!]$). Simiarly, any context that can accept outputs satisfying C's postcondition Q can also accept outputs satisfying C''s postcondition Q since $Q' \Rightarrow Q$ ($[\![Q']\!] \subseteq [\![Q]\!]$) From the point of view of sub-programs that might implement κ and κ', if κ' refines κ then every implementation that satisfies κ' also satisfies κ.

2.2.3 *Tool Foundations.* Early tools associated with Floyd/Hoare Logic methods 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 discussion of "weakest" above, 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, such as the rule for assignments

$$wp(x := E,\ Q)\ =\ Q[E/x]$$

i.e., for Q (viewed as a predicate that states a property of x) to hold after the assignment of E to x, Q should hold for the value E before the assignment. Similar rules exist for computing weakest preconditions for conditional statements [Dijkstra 1976] and manipulation of basic data structures such as records and arrays [Hoare 1972]. Due to basic undecidability results, an algorithm for calculating weakest preconditions for while loops that can be discharged by automated theorem provers is infeasible. Useful automated approaches that find or over-approximate fixpoints 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 Draconian approach of sacrificing soundness by only verifying a bounded number of loop iterations.

2.2.4 *Issues*

—Design-by-contract

2.3 Loops and Recursion

Reasoning about loops and recursion is often challenging because one must come up with 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.

When reasoning about loops, such properties are called *loop invariants*. A loop invariant is a store predicate that is always holds (*i.e.*, is invariant) at the beginning and end of each loop iteration. 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 some that the loop concludes with a set of variables satisfying P. When the loop concludes, we know that its test b must be false as well, so an appropriate postcondition is $P \wedge \neg b$. To establish this postcondition, we need to show that P holds no matter how many times the loop is executed – and this follows by showing P to be an invariant. First, because the loop body may never execute in the case where the test b is false initially, we need P to hold as a precondition to executing the **while** – this corresponds to the base case of an inductive argument. Next, we need to show that P is preserved by each loop iteration – this corresponds to an induction step. Specifically, we assume that

```
package The_Stack
--# own State : Stack_Type;   -- abstract variable
--# initializes State;
is
   --# type Stack_Type is abstract;

   --# function Not_Full(S : Stack_Type) return Boolean;
   --# function Not_Empty(S : Stack_Type) return Boolean;
   --# function Append(S : Stack_Type; X : Integer) return Stack_Type;

   procedure Push(X : in Integer);
   --# global in out State;
   --# derives State from State, X;
   --# pre Not_Full(State);
   --# post State = Append(State~, X);

   procedure Pop(X : out Integer);
   --# global in out State;
   --# derives State, X from State;
   --# pre Not_Empty(State);
   --# post State~ = Append(State, X);

end The_Stack;
```

Fig. 2. A SPARK package specification of a stack

$P \wedge b$ holds at the beginning of the loop body and we show that P holds after executing the body.

This reasoning approach can be applied to other forms of loops as illustrated by the SPARK function `Value_Present` of Figure 1. The contract for this function has a default precondition of *true* and the postcondition requires that the return value is *true* if and only there is an index `M` for array `S` such that the value in `S` at position `M` is equal to the input parameter `X`. In SPARK, loop invariants are written as assertions and can be placed anywhere in the body of the loop such that each path along a complete repetition of the loop passes through the assertion. In `Value_Present`, if `S(1) = X`, then the loop is exited before control passes through the position of the loop invariant; the invariant is not needed to establish the postcondition. If not `S(1) = X`, then iteration of the loop continues, and the loop invariant provides information that the value of `X` does not occur in the portion of the array explored thus far; this information is necessary to establish the postcondition when the of `X` is not found in the array.

JH comment: The following issues remain: Loop invariants and variant functions, and how these are affected by `break`, `continue`, and the presence of side effects in the loop test. Measure functions for recursions. Contrast VDM's use of (well-founded) relations.

2.4 Module Interfaces

Structuring constructs such as modules (*e.g.*, in Modula-3), packages (*e.g.*, in Ada), and classes (*e.g.*, in Java and C++) are used to aggregate subprograms and state that have related functionality. In our discussions, we will adopt the term *module* as a generic term for such constructs. Well-designed programming languages include features that support *information hiding*. These features allow developers to expose subprogram type-signatures and portions module state to clients while hid-

```
package body The_Stack
--# own State is S, Pointer;  -- refinement definition
is
   Stack_Size : constant := 100;
   type Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range is Pointer_Range range 1..Stack_Size;
   type Vector is array(Index_Range) of Integer;
   S : Vector;
   Pointer : Pointer_Range;

   procedure Push(X : in Integer)
   --# global in out S, Pointer;
   --# derives S       from S, Pointer, X &
   --#         Pointer from Pointer;
   --# pre Pointer < Stack_Size;
   --# post Pointer = Pointer~ + 1 and
   --#      S = S~[Pointer => X];
   is
   begin
      Pointer := Pointer + 1;
      S(Pointer) := X;
   end Push;

   procedure Pop(X : out Integer)
   --# global in S; in out Pointer;
   --# derives Pointer from Pointer &
   --#         X       from S, Pointer;
   --# pre Pointer /= 0;
   --# post Pointer = Pointer~ - 1 and
   --#      X = S(Pointer~);
   is
   begin
      X := S(Pointer);
      Pointer := Pointer - 1;
   end Pop;

begin  -- initialization
   Pointer := 0;
   S := Vector'(Index_Range => 0);
end The_Stack;
```

Fig. 3. A SPARK package body implementing a stack

ing implementation details upon which module clients should not depend [Parnas 1972].

For example, Ada provides language features to separate module definitions (called *packages* in Ada) into *package specifications* that are visible to clients and *package bodies* that are not visible to clients. Figure 2 displays an Ada package specification supplemented with SPARK annotations for a simple stack data structure. The Ada code provides types signatures for the `Push` and `Pop` — the package subprograms that are visible to clients. An Ada package specification can also provide types and variables that are visible to clients (this is not illustrated in the `Stack` example). Figure 3 displays an Ada package body that includes definitions for constants, types, variables, and procedure implementations that are hidden from clients.

The Ada features that clearly separate module interfaces from implementations improve upon many other languages that omit such features. The type signatures in package specifications already provide a simple form of functional behavior specification. SPARK supplements Ada package specifications and bodies with a number

of other forms of behavioral information. The `Stack` implementation in (Figure 3) includes variables `S` and `Pointer` as pieces of local state that are hidden from clients. Constructing detailed specifications of module functional behavior while avoiding exposure of implementation details that should remain hidden is one of the challenging aspects of module interface behavioral specification. This is challenging because behavioral specifications are often most naturally and most easily specified in terms of the concrete data structures present in the module implementation. A key aspect of module interface specification language design is the inclusion of flexible abstraction mechanisms that allow specifiers to build appropriate abstract data elements and functional specifications that can be related (through some refinement notion) to concrete data and functional behavior that must be hidden from module clients.

To enable reasoning about `Stack` behavior while hiding details of implementation, it is necessary to expose the fact that state does exist in the module implementation and that procedures in the public interface of `Stack` may change the hidden state. A SPARK *own variable* declaration (top of (Figure 2) indicates that the package contains hidden state. The single own variable named `State` abstracts the two variables `S` and `Pointer` in the package body. The SPARK `initializes` annotation in the package specification indicates that the package body (in an initialization block – bottom of Figure 3 provides the initial value of each of the hidden state variables. The next section of SPARK annotations in Figure 2 declares that the type of the own variables is **`abstract`** (not defined in the specification).

GTL comment: Need to relate the above to other languages, *e.g.*, model fields in JML. This should be a survey of concepts with SPARK used to illustrate, not just an explanation of SPARK.

The foundation of SPARK subprogram behavioral specifications are declarations that specify *frame conditions* – indicating what portions of the program state are read or written by each subprogram. Each SPARK procedure obviously may reference or update the state associated with its parameters. SPARK procedure contracts also use **`global`** annotations to indicate the global variables that the procedure references or updates. Moreover, each procedure parameter and global must be annotated with a *mode* **`in`**, **`out`**, or **`in out`** to indicate if a variable is read-only, write-only, or may be both read and written, respectively. For example, on the `Push` procedure, the input parameter `X` is read only, and the only global state that the procedure touches in the state abstracted by the own variable `State`. Together, the mode annotations on parameters and globals specify the inputs and outputs of the procedure. Frame conditions limit the scope of program state that must be mentioned in a subprograms specification; if a variable is not mentioned by a subprogram's frame condition specification, one may assume that the variable has the same value before and after calls to the subprogram and that differences in the variable's value have no effect on the subprograms behavior. We discuss frame condtions in more detail in Section 4.1.

The next step in specifying a subprogram's functional behavior in SPARK involves using SPARK **`derives`** annotations to specify how information flows from inputs to outputs – this provides a simple abstraction of the program's functional computation. The derives annotations on the `Push` procedure specify that the fi-

nal (post-state) value of the output parameter `State` is derived from the initial (pre-state) values of the input parameters `X` and `State`. SPARK information flow relations where originally proposed by Bergeretti and Carré [?] to assess the degree of coupling between program components, but they have recently used as part of a broader framework for developing certified information assurance applications [?]. In addition to checking that a procedure implementation's information flow conforms to the contract information given by **derives** clauses, the SPARK Examiner, the static analysis tool associated with SPARK, checks for a variety of simple information flow properties useful for high assurance contexts. These include: all **out** variables must be assigned along each control-flow path in the procedure, variables must be assigned before they are referenced, each **in** variable must be referenced in the procedure body, etc.

GTL comment: Need to relate the above to other languages and literature.

Figure 2 illustrates procedure contracts for `Push` and `Pop` that include pre/post-conditions stated in terms of the `State` data abstraction. To capture the functionality of the `Push`/`Pop` procedures without referring to the hidden implementation state, specification functions `Not_Full` `Not_Empty`, and `Append`. Figure 3 illustrates contracts for corresponding procedure implementations. The relationship between package specification contracts and package body contracts are discussed in Section 2.6.

2.5 Module Invariants

Single-state predicates that must hold for all states, or all states visible to clients, typically relating states of the module's variables.

Hidden vs. public invariants.

2.6 Data Abstraction

Section 2.4 discussed the need to keep module implementation details hidden as module interface specification are constructed. Data abstraction plays a key role in enabling specifications to be stated independently of concrete data in implementations.

Figure 2 illustrated SPARK's notion of **own** variable to define a specification variable `State` that abstracts the concrete variables used to represent a stack. The top of Figure 3 illustrates a SPARK data refinement definition specifying that the abstract own variable state is refined to the concrete variables `S` and `Pointer`.

The contracts for each procedure in the package body must now be restated in terms of the concrete variables that form the hidden implementation of the stack. For **derives** clauses, the SPARK Examiner checks that each **derives** clause of a procedure implementation refines the **derives** clause of the corresponding procedure specification according to **own** variable refinement definition. For example, in the `Push` procedure implementation, since `S` derives from `S`, `Pointer`, `X` applying the abstraction function corresponding to the inverse of **own** variable refinement (which would map both `S` and `Pointer` to `State`) yields an abstract refinement clause stating that `State` derives from `State` and `X`. As expected, the abstraction step introduces imprecision: declaring `State` derives from `State` and `X` admits the possibility of,

e.g., `Pointer` deriving from `S` and `X`, but this information flow relationship is not present in the procedure implementation contract.

The pre/postconditions of the procedure implementations also refer to concrete data representation. The SPARK Examiner cannot automatically verify the correspondence between the abstract specification and concrete implementation pre/-postconditions; the correspondence must be proved manually in the SPARK Proof Checker. The definition of the abstract functions `Not_Full`, `Not_Empty`, and `Append` must be specified via axioms in the proof checker language.

GTL comment: We should relate the above to the literature...

2.7 Enabling Modular/Compositional Reasoning

Explaining how many of the concepts above come together to enable compositional reasoning.

Problems, including re-entrance.

Solutions: prohibiting re-entrance (as in Anna or Spark/Ada), or allowing this with specification constructs such as "inv" in Spec#, valid() functions (in ESC/-Modula-3), dynamic frames, and in Parkinson's work.

3. EXCEPTIONAL BEHAVIOR

An *exception* is a way of returning from a function (or procedure, method) that is different than the normal return. This corresponds to throwing an exception in Java, C++, or C#. Exceptions of this type are handled by try-catch statements in Java.

We can distinguish three kinds of exceptional behavior (cf., [Goodenough 1975; Leino and Schulte 2004]):

(1) Problems of the underlying runtime environment, for instance, out-of-memory errors, stack overflow, type errors (caused by failing to recompile code after changes). In Java, these exceptions are of type `Error`. These problems can typically not be foreseen or controlled by the program. Dealing with such issues is a matter of robustness (rather than correctness), and not in the scope of this paper.

(2) Unsatisfied preconditions of operations, for instance, null-pointer exceptions, index-out-of-bounds errors, illegal casts, arithmetic overflow, division by zero, etc. In Java, these exceptions are of type `RuntimeException`. Since almost all operations in a program are partial, these expections can occur at almost any program instruction. They are typically considered errors in the program.

(3) Foreseeable problems with operations that do not always work, for instance, I/O, table look-up, etc. These exceptions are used instead of special return values to signal an unsuccessful operation.

Most specification languages ignore category 1 because it is beyond the semantics of the program (*e.g.*, Java, JML, Spec#). That is, any operation may throw an exception of this category, without explicit permission to do so. Corresponding verification systems do not impose proof obligations to prevent this kind of exception [Poetzsch-Heffter 1997; Leino and Schulte 2004].

Different languages interpret category 2 differently. For instance, in JML, a method that throws a `NullPointerException` is not necessarily incorrect. It depends on the specification of the method whether or not this kind of exception is permitted. By contrast, Spec# considers exceptions of this category as errors and imposes proof obligations to prevent them.

Exceptions of category 3 are permitted by most languages. Some languages such as Java and Spec# require them to be declared as part of the method signature, whereas others such as C# and Eiffel do not.

Languages that view exceptions as one possible method result usually permit ways to specify when this result may occur and what may be assumed in case it actually does. The most basic form of these specifications contains exceptional pre- and postconditions (*e.g.*, in Spec#). JML permits whole exceptional specification cases including assignable clauses, etc. Eiffel does not permit the specification of exceptional behavior.

Reasoning about exceptions is challenging for several reasons. First, exceptions of category 1 and 2 may occur (almost) anytime, so considering all the possible control flows leads to an extremely high complexity. Second, in the presence of side effects, it is usually unclear what an exception handler may assume about the state of the program (see JacobsMuellerPiessens07), for instance, which invariants may be assumed to hold. For checked exceptions, the exceptional postcondition provides this information, but for unchecked exceptions, one may only assume properties that hold throughout the code that potentially throws an exception (*e.g.*, the try block) and on properties the exception handler can test at runtime. So very often, there is no safe way to recover from an unchecked exception.

In the descriptions of how to specify each kind of property, we will:

—Show an example to facilitate comparisons (ideally the same example in various languages).

—Describe ways (mention tools) to check/verify such specifications.

—Mention or describe the underlying theory (first order logic, set theory, etc.), if there is any.

—Refer to any major case studies.

3.1 What to Specify

When to specify exceptional behavior vs. using preconditions to rule out exceptions.

3.2 Specification Constructs for Exceptions

Ways to specify that exceptions must or may be thrown.

Specification cases for specifying different behaviors.

4. HEAP MANIPULATION

In order to handle heap data structures, a specification technique must be able to describe (a) the topology of a data structure and (b) the effects that methods have on heap data structures. We illustrate these two aspects 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 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:

—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, for instance, needed if methods are allowed to occur in contracts like in Eiffel, JML, and Spec#.

—Allocation and de-allocation effects: which objects are allocated or de-allocated by a method. Information about allocation is, for instance, needed to prove that the result of a method is different from all existing objects. De-allocation 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.

challenges: abstraction (information hiding, subclasses) support reasoning (logics) strong support for 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. Hoare's original invariance rule illustrates framing

$$\frac{\{P\}\,C\,\{Q\}}{\{P \wedge R\}\,C\,\{Q \wedge R\}} \text{ provided effects of } C \text{ are disjoint from } \mathrm{FV}(R)$$

Here, R is the frame. The rule says that an assertion about some state that is unmodified by C can be preserved. However, with the addition of the heap the sidecondition 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 shapes that C changes are disjoint from the shapes that R depends upon.

The existing approaches can be grouped into three major categories (for specifying topology):

Fig. 4.

(1) Approaches based on predicate logic. Languages in this category specifiy heap manipulation in terms of a heap model that is given in predicate logic. Examples are dynamic frames and regional logic (the above paper by Kuncak et al. also falls into this category.) Structures are described by defining functions or regions and using them to express disjointness, inclusion, etc. of object structures. Framing is done by showing disjointness of read and write effects.

(2) Approaches based on specialized logics. Languages in this category are based on logics with special primitives for heap properties. The most important representative of this category is separation logic with its separating conjunction. Separating conjunction can express disjointness of structures and, therefore, also acyclicity. Framing is done with a special frame rule.

(3) Approaches based on type systes. Languages in this category use types to describe and check heap structures. The most prominent representative of this category is ownership. The described structures are hierchies of objects with certain encapsulation policies. Framing is usually done by specifying read and write effects in terms of ownership trees. There are other candidates in this category such as alias types by Morrisett et al.

There are several approaches that are hybrids in the above classification. For instance, Spec# describes structures and applies framing using ownership (3), but encodes ownership in a heap model in predicate logic (1). Implicit dynamic frames by Smans et al. combine 1 and 2. The above paper by Moller and Schwartzbach seems to combine 2 and 3 (graph types and assertions containing routing expressions).

MJP comment: Some thoughts

Properties of a linked list based stack

(1) It is a list (2) Its values are non-null object-references. (3) How do we describe two stacks are disjoint (4) The same object reference can be stored in two stacks.

Initial Outline:

In the descriptions of how to specify each kind of property, we will:

—Show an example to facilitate comparisons (ideally the same example in various languages).

—Describe ways (mention tools) to check/verify such specifications.

—Mention or describe the underlying theory (first order logic, set theory, etc.)

—Refer to any major case studies.

4.1 Framing

Notations for frame conditions, specification of side effects on arguments, etc.

4.2 Alias Control, Ownership, and Separation

Notations for controlling or limiting aliasing (e.g., ownership types).

Ideas behind separation logic. Concepts and basic notations.

How to use separation logic in specifications.

Comparison of separation logic and other notations for framing, alias control, and ownership. (such as a specialized type system, like an ownership type system, the Kassios dynamic-frames approach or the Boogie methodology used in Spec#).

5. OBJECT ORIENTATION

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* is a kind of constrained, ad-hoc polymorphism [Wadler and Blott 1989]. 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 published an influential study of the conditions for type checking OOP [Cardelli 1988]. Cardelli's subtyping rules prevent type errors, and can be extended to languages with more features, such as multiple dispatch [Castagna 1995].

Dynamic dispatch is a semantic feature of OO langauges 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 or inherited by `coll`'s static type, `Collection`, or may be an overriding 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. This use of subtype polymorphism is at the core of most object-oriented design patterns [Gamma et al. 1995].

However, it is not only the implementation to be executed that is not known statically (in general), but without some methodological convention, such as behavioral subtyping, even the specification of the method that will be executed will not be known statically. This presents a problem for static verification. One approach to solving this probem is to describe what a method call does for each possible subtype of the receiver's type. However, such a case analysis would be difficult to maintain, as each time a new subtype was 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 *behavioral subtyping* [America 1987; Bruce and Wegner 1990; Dhara and Leavens

1996; Leavens and Dhara 2000; Leavens and Weihl 1995; 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. Put the other way around, 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]. Just as a type system that ensures subtyping ensures that no runtime type errors occur when calling methods using dynamic dispatch, using behavioral subtyping ensures that no surprising behavior occurs when calling methods using dynamic dispatch.

A simple way of enforcing behavioral subtyping is to impose the following three 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 frame of $D.m$ must be a subset of the frame of $C.m$. 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 also the precondition of the method selected at runtime, $D.m$, 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 leaves the value of some location x unchanged, then $D.m$ also does. 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 to handle 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. 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 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 the above 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 only be called 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 is stronger than the C-invariant). These two properties allow for weaker versions of rules 1–3. For each instance method $D.m$ that overrides a method $C.m$:

(1) 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.*
(2) 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.*
(3) 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, the weaker version of rule 3 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 to assume 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 [Ádám Darvas and Müller 2005]. For instance, the possible values of a model field may be restriced in subclasses. Thus, more precise type information for the receiver provides more information about the values of its model fields, which may help to prove all four of the above rules.

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 also held in the call's pre-state [Dhara and Leavens 1996; Chen and Cheng 2000]:

(2) 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* .
(3) 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: 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. 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 that the effective invariant is stronger than the invariants of D's supertypes and thus that rule 4 is satisfied.

Specification inheritance enforces behavioral subtyping [Dhara and Leavens 1996]. However, the above rules have been criticized to disguise specification errors because failure to comply with the rules of behavioral subtyping is silently turned into bogus specifications. Assume that an overridden supertype method has the declared precondition `p > 0` and the overriding subtype method requires `p <= 0`. With specification inheritance, however, the effective precondition of the overriding method is `true`. Findler and Felleisen [Findler and Felleisen 2001] argue that this is most likely 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.

Despite this criticism, most existing specification languages enforce behavioral subtyping through specification inheritance. JML [Leavens 2006; Leavens et al. 2008] defines effective preconditions and invariants as described above. For postconditions, JML exploits the second weakening of behavioral subtyping rule 3: 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 $pre_c \Rightarrow post_c$ rather than a conjunction of postconditions $post_c$. The rule for frame properties similary depends on which precondition holds. 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 $\bigwedge_c (pre_c \Rightarrow post_c) \wedge post_D$, 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 qualification by preconditions 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.

[[Peter: I would have liked to illustrate the concepts with an example, but I did not manage to come up with a natural example. And artificial examples do not help much.]]

[[Peter: I did not show formulas because I did not want to introduce all the required notations, but maybe we have to make the rules clear.]]

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]. 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; dynamic specifications are used to reason about dynamically-dispatched calls.

6. CONCLUSIONS

Discuss the challenge of interactions between kinds of properties, such as between functional (data) and sequencing (temporal logic) properties, and between functional properties and resource properties.

Collect accomplishments and key ideas useful for the grand challenge.

6.1 Future Work

Describe research problems that need to be addressed in the context of the grand challenge.

REFERENCES

Abadi, M. and Lamport, L. 1988. The existence of refinement mappings. Tech. Rep. 29, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301. Aug. A shorter version appeared in *Proceedings of the LICS Conference*, Edinburgh, Scotland, July 1988.

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

Ádám Darvas and Müller, P. 2005. Reasoning about method calls in JML specifications. In *Formal Techniques for Java-like Programs*. ETH, Zurich, Switzerland.

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

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, New York, NY, 414–425.

Alur, R. and Henzinger, T. A. 1992. Logics and models of real time: A survey. In *Real Time: Theory in Practice*, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, Eds. Lecture Notes in Computer Science, vol. 600. Springer-Verlag, New York, NY, 74–106.

Alur, R. and Henzinger, T. A. 1994. A really temporal logic. *Journal of the 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. *ACM SIGPLAN Notices 12,* 3 (Mar.), 1–10. Proceedings of the ACM Conference on Language Design for Reliable Software.

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

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 (Oct.), 431–483.

Apt, K. R. and Olderog, 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. ed. Graduate texts in computer science series. Springer-Verlag, New York, NY.

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

Astesiano, E. 1991. Inductive and operational semantics. In *Formal Description of Programming Concepts*, E. J. Neuhold and M. Paul, Eds. IFIP State-of-the-Art Reports. Springer-Verlag, New York, NY, 51–136.

Back, R.-J. R. 1978. On the correctness of refinement steps in program development. Ph.D. thesis, University of Helsinki. Report A-1978-4.

Backhouse, R., Chisholm, P., Malcolm, G., and Saaman, E. 1989. Do-it-yourself type theory. *Formal Aspects of Computing 1,* 1 (January – March), 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.

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

Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Everett, B., and Cooper, D. 2006. Engineering the tokeneer enclave protection system. In *IEEE International Symposium on Secure Software Engineering.* IEEE, Los Alamitos, California.

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., 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.

Bellini, P., Mattolini, R., and Nesi, R. 2000. Temporal logics for real-time system specification. *ACM Computing Surveys 32,* 1 (Mar.), 12–42.

Bernot, G., Claudel, M. C., and Marre, B. 1991. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal 6,* 6 (Nov.), 387–405.

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, Mass.

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. ISBN 0-387-54060-1.

Bjørner, D. and Henson, M. C. 2008. *Logics of Specification Languages.* Springer-Verlag, Berlin.

Boehm, H.-J. 1985. Side effects and aliasing can have simple axiomatic descriptions. *ACM Trans. Program. Lang. Syst. 7,* 4 (Oct.), 637–655.

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

Brookes, S. D., Hoare, C. A. R., and Roscoe, A. W. 1984. A theory of communicating sequential processes. *Journal of the ACM 31,* 3 (July), 560–599.

Brooks, Jr., F. P. 1987. No silver bullet: Essence and accidents of software engineering. *Computer 20,* 4 (Apr.), 10–19.

Bruce, K. B. and Wegner, P. 1990. An algebraic model of subtype and inheritance. In *Advances in Database Programming Languages*, F. Bançilhon and P. Buneman, Eds. Addison-Wesley, Reading, Mass., 75–96.

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. *International Journal on Software Tools for Technology Transfer 7,* 3 (June), 212–232.

Cansell, D. and Méry, D. 2008. The event-B modelling method: Concepts and case studies. See Bjørner and Henson [2008], 47–152.

Cardelli, L. 1988. A semantics of multiple inheritance. *Information and Computation 76,* 2/3 (February/March), 138–164.

Cartwright, R. and Felleisen, M. 1996. Program verification through soft typing. *ACM Comput. Surv. 28,* 2 (June), 349–351.

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

Chang, J., Richardson, D. J., and Sankar, S. 1996. Structural specification-based testing with ADL. In *Proceedings of ISSTA 96, San Diego, CA*. IEEE Computer Society, Los Alamitos, California, 62–70.

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. 1994a. The Larch/Smalltalk interface specification language. *ACM Transactions on Software Engineering and Methodology 3,* 3 (July), 221–253.

Cheon, Y. and Leavens, G. T. 1994b. A quick overview of Larch/C++. *Journal of Object-Oriented Programming 7,* 6 (Oct.), 39–49.

Cheon, Y. and Leavens, G. T. 2002. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, B. Magnusson, Ed. Lecture Notes in Computer Science, vol. 2374. Springer-Verlag, Berlin, 231–255.

Chin, W.-N., David, C., Nguyen, H. H., and Qin, S. 2008. Enhancing modular oo verification with separation logic. In *ACM Symposium on Principles of Programming Languages*, P. Wadler, Ed. ACM, New York, NY, 87–99.

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, L. A. and Rosenblum, D. S. 2006. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes 31,* 3 (Mar.), 25–37. http://doi.acm.org/10.1145/1127878.1127900.

CoFI (The Common Framework Initiative). 2004. Casl *Reference Manual.* LNCS 2960 (IFIP Series). Springer-Verlag, Berlin.

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, Chapter 15, 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 *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages.* ACM, 238–252.

Crowley, J. L., Leathrum, J. F., and Liburdy, K. A. 1996. Issues in the full scale use of formal methods for automated testing. *ACM SIGSOFT Software Engineering Notes 21,* 3 (May), 71–78.

Detlefs, D. L., Leino, K. R. M., Nelson, G., and Saxe, J. B. 1998. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto. Dec.

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

Dijkstra, E. W. 1976. *A Discipline of Programming.* Prentice-Hall, Inc., Englewood Cliffs, N.J.

Dong, J. S., Hao, P., Zhang, X., and Qin, S. C. 2006. Highspec: a tool for building and checking ozta models. In *ICSE '06: Proceedings of the 28th international conference on Software engineering.* ACM, New York, NY, USA, 775–778.

Edmund M. Clarke, J., Grumberg, O., and Peled, D. A. 1999. *Model Checking.* The MIT Press, Cambridge, Mass.

Edwards, S. H., Heym, W. D., Long, T. J., Sitaraman, M., and Weide, B. W. 1994. Part II: Specifying components in RESOLVE. *ACM SIGSOFT Software Engineering Notes 19,* 4 (Oct.), 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, Mass., Chapter 16, 995–1072.

Ernst, M. D. 2003. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis, Portland, OR.* Jonathan Cook, New Mexico State University, 24–27.

Filliâtre, J.-C. 2003. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud. Mar.

Findler, R. B. and Felleisen, M. 2001. Contract soundness for object-oriented languages. In *OOPSLA '01 Conference Proceedings, Object-Oriented Programming, Systems, Languages, and Applications, October 14-18, 2001, Tampa Bay, Florida, USA.* 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. See Bjørner and Henson [2008], 453–487.

Fitzgerald, J. and Larsen, P. G. 1998. *Modelling Systems: Practical Tools in Software Development.* Cambridge, Cambridge, UK.

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).* ACM SIGPLAN Notices, vol. 41, 1. ACM Press, New York, 245–256.

Floyd, R. W. 1967. Assigning meanings to programs. *Proceedings Symposium on Applied Mathematics 19*, 19–31.

Francez, N. 1992. *Program Verification.* Addison-Wesley Publishing Co., Cambridge, UK.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Mass.

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, Chapter 3, 47–67.

Geller, M. 1978. Test data as an aid in proving program correctness. *Commun. ACM 21,* 5 (May), 368–375.

George, C. and Haxthausen, A. E. 2008. The logic of the RAISE specification language. See Bjørner and Henson [2008], 349–399.

German, S. M. 1978. Automating proofs of the absence of common runtime errors. In *Conference record of the Fifth Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 105–118.

Goguen, J. A. and Malcolm, G. 1996. *Algebraic Semantics of Imperative Programs*. MIT Press, Cambridge, MA.

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*, R. T. Yeh, Ed. Vol. 4. Prentice-Hall, Inc., Englewood Cliffs, N.J., 80–149.

Goodenough, J. B. 1975. Structured exception handling. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages*. ACM, 204–224.

Gurevich, Y. 1991. Evolving algebras: A tutorial introduction. *Bulletin of the EATCS 43*, 264–284.

Guttag, J. and Horning, J. J. 1978. The algebraic specification of abstract data types. *Acta Informatica 10,* 1, 27–52.

Guttag, J. V. and Horning, J. J. 1986. Report on the Larch Shared Language. *Sci. Comput. Program. 6,* 2 (Mar.), 103–134.

Guttag, J. V. and Horning, J. J. 1991. A tutorial on Larch and LCL, a Larch/C interface language. In *VDM '91 Formal Software Development Methods 4th International Symposium of VDM Europe Noordwijkerhout, The Netherlands, Volume 2: Tutorials*, S. Prehn and W. J. Toetenel, Eds. Lecture Notes in Computer Science, vol. 552. Springer-Verlag, New York, NY, 1–78.

Guttag, J. V., Horning, J. J., Garland, S., Jones, K., Modet, A., and Wing, J. 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 (Sept.), 24–36.

Halbwachs, N. 2005. A synchronous language at work: the story of Lustre. In *MEMOCODE*. IEEE, Los Alamitos, CA, 3–11.

Halbwachs, N., Lagnier, F., and Ratel, C. 1992. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering 18,* 9 (Sept.), 785–793.

Hall, A. and Chapman, R. 2002. Correctness by construction: Developing a commercial secure system. *IEEE Software 19*, 18–25.

Hansen, M. R. 2008. Duration calculus. See Bjørner and Henson [2008], 299–347.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program. 8,* 3 (June), 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 Transactions on Software Engineering 16,* 4 (Apr.), 403–413.

Harms, D. E. and Weide, B. W. 1991. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering 17,* 5 (May), 424–435.

Hayes, I., Ed. 1993. *Specification Case Studies*, Second ed. International Series in Computer Science. Prentice-Hall, Inc., London.

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

Heimdahl, M. P. E., Whalen, M. W., and Thompson, J. M. 2003. NIMBUS: A tool for specification centered development. In *RE '03: Proceedings of the 11th IEEE International Conference on Requirements Engineering*. IEEE Computer Society, Washington, DC, USA, 349.

Heitmeyer, C., Jeffords, R., Bharadwaj, R., and Archer, M. 2007. Re theory meets software practice: Lessons from the software development trenches. In *15th IEEE International Requirements Engineering Conference, 2007 (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 Transactions on Software Engineering 24,* 11 (Nov.), 927–948.

Heitmeyer, C. L. and Jeffords, R. D. 2007. Applying a formal requirements method to three NASA systems: Lessons learned. In *Proceedings of the 2007 IEEE Aerospace Conference*. IEEE Computer Society Press, Los Alamitos, California.

Hennessy, M. 1990. *The Semantics of Programming Languages: an Elementary Introduction using Structural Operational Semantics*. John Wiley and Sons, New York, NY.

Hennessy, M. and Milner, R. 1985. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM 32,* 1 (Jan.), 137–161.

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

Hoare, C. A. R. 1972. Notes on data structuring. In *Structured Programming*, O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. Academic Press, Inc., New York, NY, 83–174.

Hoare, C. A. R. 1985. *Communicating Sequential Processes*. Prentice-Hall, Inc., Englewood Cliffs, N.J.

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

Hoare, C. A. R., Hayes, I. J., Jifeng, H., Morgan, C. C., Roscoe, A. W., Sanders, J. W., Sorensen, I. H., Spivey, J. M., and Sufrin, B. A. 1987. Laws of programming. *Commun. ACM 30,* 8 (Aug.), 672–686. See corrections in the September 1987 CACM.

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 Transactions on Software Engineering 23,* 5 (May), 279–295.

Jackson, D. 1995. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology 4,* 4 (Oct.), 365–389.

Jackson, D. 2006. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass.

Jackson, D. and Rinard, M. 2000. Software analysis: A roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*. ACM, New York, NY, USA, 133–145.

Jahanian, F. and Mok, A. K. 1986. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering 12,* 9, 890–904.

Jalote, P. 1992. Specification and testing of abstract data types. *Computing Languages 17,* 1, 75–82.

Jeffords, R. and Heitmeyer, C. 1998. Automatic generation of state invariants from requirements specifications. *ACM SIGSOFT Software Engineering Notes 23,* 6 (Nov.), 56–69. Proceedings of the ACM SIGSOFT Sixth Internatioal Symposium on the Foundations of Software Engineering.

Jones, C. B. 1990. *Systematic Software Development Using VDM*, Second ed. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J.

Jones, C. B. 2003. The early search for tractable ways of reasoning about programs. *IEEE Annals of the History of Computing 25,* 2, 26–49.

Jones, K. D. 1991. LM3: A Larch interface language for Modula-3: A definition and introduction: Version 1.0. Tech. Rep. 72, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue Palo Alto, CA 94301. June. Order from src-report@src.dec.com.

King, S., Hammond, J., Chapman, R., and Pryor, A. 1999. The value of verification: Positive experience of industrial proof. In *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings*, J. M. Wing, J. Woodcock, and J. Davies, Eds. Number 1709 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1527–1545.

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.

Korel, B. and Al-Yami, A. M. 1998. Automated regression test generation. *ACM SIGSOFT Software Engineering Notes 23,* 2 (Mar.), 143–152. ISSTA 98: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis.

Kozen, D. and Tiuryn, J. 1990. Logics of programs. In *Handbook of Theoretical Computer Science*, J. van Leewen, Ed. Vol. B: Formal Models and Semantics. The MIT Press, New York, NY, Chapter 14, 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 (Jan.), 32–45.

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

Lampson, B. and Burstall, R. 1988. Pebble, a kernel language for modules and abstract data types. *Information and Computation 76,* 2/3 (February/March), 278–346. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 1–50.

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. Oct. Also *SIGPLAN Notices*, 12(2), February, 1977.

Lano, K. 1996. *The B Language and Method: A guide to Practical Formal Development.* Formal Appoaches to Computing and Information Technology. Springer-Verlag, London, UK.

Larsen, K. G., Pettersson, P., and Yi, W. 1997. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer 1,* 1-2 (Oct.), 134–152.

Leavens, G. T. 2006. JML's rich, inherited specifications for behavioral subtypes. In *Formal Methods and Software Engineering: 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 Software Engineering Notes 31,* 3 (Mar.), 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. *Science of Computer Programming 55,* 1-3 (Mar.), 185–208.

Leavens, G. T. and Dhara, K. K. 2000. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, Cambridge, UK, Chapter 6, 113–135.

Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D. R., Müller, P., Kiniry, J., Chalin, P., and Zimmerman, D. M. 2008. JML Reference Manual. Available from `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 (Nov.), 705–778.

Leino, K. R. M. 1995. Toward reliable modular programs. Ph.D. thesis, California Institute of Technology. Available as Technical Report Caltech-CS-TR-95-03.

Leino, K. R. M. 2008. This is Boogie 2. Manuscript KRML 178. Available at `http://research.microsoft.com/~leino/papers.html`.

Leino, K. R. M. and Müller, P. 2006. A verification methodology for model fields. In *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 Schulte, W. 2004. Exception safety for C#. In *SEFM 2004—Second International Conference on Software Engineering and Formal Methods*, J. R. Cuellar and Z. Liu, Eds. IEEE, 218–227.

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 *Software Engineering – ESEC/FSE '99*, O. Nierstrasz and M. Lemoine, Eds. Lecture Notes in Computer Science, vol. 1687. Springer-Verlag, Berlin, 127–145. Also ACM SIGSOFT *Software Engineering Notes*, volume 24, number 6.

Liskov, B. 1988. Data abstraction and hierarchy. *ACM SIGPLAN Notices 23,* 5 (May), 17–34. Revised version of the keynote address given at OOPSLA '87.

Liskov, B. and Guttag, J. 1986. *Abstraction and Specification in Program Development.* The MIT Press, Cambridge, Mass.

Liskov, B. H. and Wing, J. M. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16,* 6 (Nov.), 1811–1841.

Loeckx, J. and Sieber, K. 1987. *The Foundations of Program Verification (Second edition).* John Wiley and Sons, New York, NY.

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 Software 2,* 2 (Mar.), 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.

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

Martin-Löf, P. 1985. Constructive mathematics and computer programming. In *Mathematical Logic and Programming Languages.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 167–184.

McConnell, S. 1993. *Code Complete: A practical handbook of software construction.* Microsoft Press, Buffalo, NY.

Meyer, B. 1992. Applying "design by contract". *Computer 25,* 10 (Oct.), 40–51.

Meyer, B. 1997. *Object-oriented Software Construction*, second ed. Prentice Hall, New York, NY.

Milne, G. and Milner, R. 1979. Concurrent processes and their syntax. *Journal of the ACM 26,* 2 (Apr.), 302–321.

Milner, R. 1990. Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science*, J. van Leewen, Ed. Vol. B: Formal Models and Semantics. The MIT Press, New York, NY, Chapter 19, 1201–1242.

Milner, R. 1991. The polyadic π-calculus: a tutorial. Tech. Rep. ECS–LFCS–91–180, LFCS. Oct. *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991.

Milner, R., Parrow, J., and Walker, D. 1992. A calculus of mobile processes, (Parts I and II). *Information and Computation 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, Chapter 6, 130–136.

Morgan, C. 1994. *Programming from Specifications: Second Edition.* Prentice Hall International, Hempstead, UK.

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. *Software—Practice & Experience 10,* 4 (Apr.), 249–263.

Mossakowski, T., Haxthausen, A. E., Sanella, D., and Tarlecki, A. 2008. Casl — the Common Algebraic Specification Language. See Bjørner and Henson [2008], 241–298.

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

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

Nielsen, M., Havelund, K., Wagner, K. R., and George, C. 1989. The raise language, method and tools. *Formal Aspects of computing 1,* 1 (Jan-Mar), 85–114.

Nielson. 1996. Annotated type and effect systems. *ACM Comput. Surv. 28,* 2 (June), 344–345.

Object Management Group. 1992. *The Common Object Request Broker: Architecture and Specification*, 1.1 ed. Object Management Group, Inc., 492 Old Connecticut Path, Framingham, MA 01701.

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 Software Engineering Notes 19,* 4 (Oct.), 23–28.

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 *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, Washington, DC, USA, 188–197.

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

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

Partsch, H. and Steinbrüggen, R. 1983. Program transformation systems. *Computing Surveys 15,* 3 (Sept.), 199–236.

Perl, S. E. and Weihl, W. E. 1993. Performance assertion checking. In *14th ACM Symposium on Operating Systems Principles.* ACM, New York, NY, 134–145.

Peters, D. K. and Parnas, D. L. 1998. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering 24,* 3 (Mar.), 161–173.

Peterson, J. L. 1977. Petri nets. *ACM Comput. Surv. 9,* 3 (Sept.), 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 (14th POPL 1987).* ACM, New York, NY, 178–188.

Plotkin, G. 1981. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Aarhus University. Sept.

Plotkin, G. D. 1977. LCF considered as a programming language. *Theor. Comput. Sci. 5*, 223–255.

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

Richardson, D. J. 1994. TAOS: Testing with analysis and oracle support. In *Proceedings of ISSTA 94, Seattle, Washington, August, 1994.* IEEE Computer Society, Los Alamitos, California, 138–152.

Roscoe, A. W. 1994. Model-checking CSP. In *A classical mind: essays in honour of C. A. R. Hoare.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, 353–378.

Rumbaugh, J., Jacobson, I., and Booch, G. 1999. *The Unified Modeling Language Refrence Manual.* Object Technology Series. Addison Wesley Longman, Reading, Mass.

Sankar, S. and Hayes, R. 1994. ADL: An interface definition language for specifying and testing software. *ACM SIGPLAN Notices 29,* 8 (Aug.), 13–21. Proceedings of the Workshop on Interface Definition Language, Jeannette M. Wing (editor), Portland, Oregon.

Schmidt, D. A. 1986. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., Boston, Mass.

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

Scott, D. S. and Strachey, C. 1971. Toward a mathematical semantics for computer languages. In *Proceedings Symposium on Computers and Automata.* Microwave Institute Symposia Series, vol. 21. Polytechnic Institute of Brooklyn, New York, NY, 19–46.

Sites, R. L. 1974. Proving that computer programs terminate cleanly. Ph.D. thesis, Stanford University. Technical Report STAN-CS-74-418.

Spivey, J. 1989. An introduction to Z and formal specifications. *Software Engineering Journal 4,* 1 (Jan.), 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.

Tan, Y. M. 1995. *Formal Specification Techniques for Engineering Modular C Programs.* Kluwer International Series in Software Engineering, vol. 1. Kluwer Academic Publishers, Boston.

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.

Wadler, P. and Blott, S. 1989. How to make ad-hoc polymorphism less ad hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas.* ACM, New York, NY, 60–76.

Wand, M. 1979. Final algebra semantics and data type extensions. *Journal of Computer and System Sciences 19,* 1 (Aug.), 27–44.

Warmer, J. and Kleppe, A. 1999. *The Object Constraint Language: Precise Modeling with UML.* Addison Wesley Longman, Reading, Mass.

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

Wing, J. M. 1990. A specifier's introduction to formal methods. *Computer 23,* 9 (Sept.), 8–24.

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

Wirsing, M. 1990. Algebraic specification. In *Handbook of Theoretical Computer Science*, J. van Leewen, Ed. Vol. B: Formal Models and Semantics. The MIT Press, New York, NY, Chapter 13, 675–788.

Zaremski, A. M. and Wing, J. M. 1997. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology 6,* 4 (Oct.), 333–369.