On Behavioral Subtyping and Completeness

Cees Pierik[1] and Frank S. de Boer[1,2,3]

[1] ICS, Utrecht University, The Netherlands
[2] CWI, Amsterdam, The Netherlands
[3] LIACS, Leiden University, The Netherlands
cees@cs.uu.nl frb@cwi.nl

Abstract. Behavioral subtyping forces objects of subtypes to behave in the same way as objects of supertypes. It is often favored over standard subtyping because it provides a means to obtain a modular program logic. Relative completeness is a formal property of a Hoare logic that ensures that any failed attempt to verify the correctness of a program is not caused by a weakness of its Hoare rules.
In this paper, we argue that the standard relative completeness notion is too stringent for program logics that are based on behavioral subtyping. Subsequently, we propose two novel and complementary completeness notions that can be employed to assess the strength of program logics that rely on behavioral subtyping.

1 Introduction

Investigating the relative completeness of a Hoare logic is the classical way to evaluate its strength [2]. Relative completeness is a formal property of a Hoare logic that ensures that any failed attempt to verify the correctness of a program is not caused by a weakness of one of its reasoning rules or axioms. Many program logics are known that satisfy this property.

We do not believe that a program logic must necessarily be complete in the above sense in order to be useful. Several logics of which the completeness is unknown have been applied in industrial studies in recent years. Moreover, extended static checkers intentionally employ incomplete logics to increase their cost-effectiveness [8, 4, 3].

However, this does not mean that completeness becomes irrelevant. For example, the documentation of an extended static checker will have to explain its limitations in detail in order to prevent its users from frantically trying to prove claims that are beyond its reach. A precise definition of completeness is a prerequisite for this task. This paper aims to give such a definition for program logics that are based on behavioral subtyping.

Behavioral subtyping is a particular stance on subtyping. It demands that what holds regarding the behavior of elements of a type also holds for elements of its subtypes [1, 13]. In other words, each type should be a refinement of its supertype.

Behavioral subtyping validates proof rules for method calls that justify the specification of a call using the specification of the corresponding method in some supertype of the actual dynamic type of the receiver. This technique is known as supertype abstraction [11]. Moreover, it also ensures that new subtypes cannot invalidate existing proofs by forcing overriding methods to satisfy the specifications of the methods that they override. Thus behavioral subtyping leads to modular proof systems, i.e., program logics with proofs that cannot be invalidated by program extensions.

Several proof logics have been proposed that are based on behavioral subtyping. However, the completeness of these logics has never been properly investigated. This situation is rather dissatisfactory because it is far from clear that these logics are complete. Behavioral subtyping raises several novel questions with respect to completeness. For example, the above-mentioned supertype abstraction principle must clearly be used with care because an implementation of the dynamic type of an object may have a stronger specification than the corresponding implementation of its supertype. It is also unclear if the specification inheritance principle [7] that is sometimes used to enforce behavioral subtyping leads to a complete proof system. These questions become even more challenging if one also imposes scoping restrictions on specifications, which is common in program logics for open programs.

In this paper, we argue that the standard relative completeness notion is too stringent for program logics which are based on behavioral subtyping. Subsequently, we propose two novel and complementary completeness notions that can be employed to assess the strength of program logics that rely on behavioral subtyping.

2 Behavioral Subtyping

This section defines behavioral subtyping in the context of a Java-like language. Our aim is to employ general descriptions of the relevant concepts. We will therefore avoid committing ourselves to some particular verification framework.

2.1 Java-like Programs

We assume that we are dealing with a Java-like programming language in which each program consists of classes. A class declaration specifies the (unique) name C of the class, its parent class D, its supertype E, a set of fields, and a set of instance methods.

$$\mathit{class} \in \mathit{Class} ::= \mathsf{class}\ C\ \mathsf{extends}\ D\ \mathsf{refines}\ E\ \{\ \mathit{field}^*\ \ \mathit{meth}^*\}$$

We use C, D, and E as typical elements of the set of class names. A clause extends D indicates that the new class is a subclass of class D, implying that it inherits the attributes of class D. We write $C \lhd D$ if C is a direct subclass of D. The subclass relation $\unlhd$ is the usual reflexive and transitive closure of the $\lhd$ relation.

A clause refines E declares the new class C to be a subtype of class E. We require that $D \unlhd E$, thus ensuring that the subclass relation subsumes the subtype relation. Declaring C to be a subtype of E implies that C-objects behave in the same way as E-objects. By distinguishing between the subclass and the subtype relation we give programmers the opportunity to reuse code using inheritance without the obligation that the new class should behave in the same way as its superclass (cf. [1]). We write $C \prec E$ if C is declared as a subtype of E, and $C \preceq E$ if (C, E) is an element of the reflexive and transitive closure of the $\prec$ relation.

Java and C# identify the subtype relation with the subclass relation and do not support a refines clause. However, it is of course possible to add this kind of information as part of the specification of a class. We will assume in this paper that programs are typechecked using the subtype relation instead of the subclass relation. For example, an assignment $x := e$ is valid if $[e] \preceq [x]$, where $[e]$ and $[x]$ denote the static types of the expression e and the variable x, respectively.

A method declaration lists the return type t of the method, its name m, a list of parameter types $t_1, \ldots, t_n$, and a statement S (its body).

$$\mathit{meth} \in \mathit{Meth} ::= t\ m(t_1, \ldots, t_n)\ \{\ S\ \}$$

A method name m should be unique in its class; we do not consider overloading for simplicity. Note that a method does not specify the identifiers of its parameters. A method with parameter types $t_1, \ldots, t_n$ is assumed to have parameters $p_1, \ldots, p_n$. We only consider public methods in this paper.

We use the following definition of overriding. Let the implementation of method m in class C have return type t and parameters types $t_1, \ldots, t_n$. Furthermore, assume that the implementation of method m in class D has return type t' and parameter types $t'_1, \ldots, t'_m$. The implementation in class D then *overrides* the implementation in class C if (1) $D \trianglelefteq C$, (2) $n = m$, (3) $t' \preceq t$, and (4) $t_i \preceq t'_i$ for every $i \in \{1 \ldots n\}$. That is, we allow contravariant changes in parameter types, and covariant changes in return types.

2.2 Hoare-like Specifications

We will use standard Hoare triples of the form $\{P\}\ S\ \{Q\}$ as partial correctness formulas of a statement S. We will not be specific about the syntax of formulas; we will only assume that the values of P and Q are either true or false in a given program state.

Secondly, we will also use Hoare triples of the form $\{P\}\ m@C\ \{Q\}$. This kind of triple specifies the behavior of a method implementation. A Hoare triple $\{P\}\ m@C\ \{Q\}$ states that the final state of every terminating computation of the implementation of method m in class C satisfies Q provided that its initial state satisfies P.

We write $\models \{P\}\ .\ \{Q\}$ if the Hoare triple $\{P\}\ .\ \{Q\}$ is valid. By $\vdash \{P\}\ .\ \{Q\}$ we denote that the Hoare triple $\{P\}\ .\ \{Q\}$ can be derived using the axioms and rules of a particular Hoare-like logic.

2.3 Behavioral Subtyping and Refinement

According to Leavens, behavioral subtyping is essentially refinement of object-types [10]. We will also stress the relationship between behavioral subtyping and refinement by defining the former in terms of a refinement relation between method implementations.

So far, we have said that a class is a behavioral subtype of another class if its objects show the same behavior as objects of its supertype. The behavior of objects is determined by the behavior of the instance methods that can be called on the objects. A method of a behavioral subtype must therefore reveal the same behavior as the corresponding method of its supertype. This requirement is trivial for methods that a class inherits from its supertype. However, behavioral subtyping imposes restrictions on methods that override methods of a supertype. In the sequel, we will refer to the methods implementations that a class C declares or inherits as the methods of C; this set does *not* include inherited method implementations which are overridden by methods that are declared in C.

Let $m@C'$ be a method of C. Let $m@D'$ be a method of class D that overrides method $m@C'$. We say that $m@D'$ *refines* the method specification $\{P\}m@C'\{Q\}$ if $\models \{P\}m@D'\{Q\}$. The latter should be checked in the context of class D, which ensures that the receiver this has type D, but the formal parameters $p_1, \ldots, p_n$ must retain the types that are specified in the declaration of method $m@C'$. One may, however, assume that the return expression has the static type that is specified in the declaration of method $m@D'$. We simply say that $m@D'$ *refines* $m@C'$ if $m@D'$ overrides method $m@D'$, and moreover, $m@D'$ refines every valid method specification $\{P\}m@C'\{Q\}$. Finally, we say that *a class D is a behavioral subtype of some class C* if class D is a subclass of class C, and moreover, each method $m@D'$ of class D that overrides a method $m@C'$ of class C also refines that method.

It is usually not necessary to check whether every class D that is declared to be a subtype of class C is indeed a behavioral subtype of class C in the given sense. It may be enough to ensure that every method of D that overrides a method of class C refines the given, fixed method specification of that method. The latter check suffices if we view method specifications as an integral part of a type, which is, e.g., done in the early work on behavioral subtyping [1, 13]. We will use the corresponding weaker definition of behavioral subtypes in Section 5.

The definition of refinement that we have given above is intentionally rather basic. In the context of open programs, one should additionally take modifies clauses and data groups into account [12]. Invariants and constraints can also be incorporated in the definition [13]. The completeness notions that we propose in this paper also make sense with more enhanced refinement notions.

3 Relative Completeness

Completeness of a logic means, in general, that every valid formula can be derived within the logic. Completeness of a Hoare logic boils down to the property that

$$\models \{P\}\ S\ \{Q\} \text{ implies } \vdash \{P\}\ S\ \{Q\}$$

for every statement S, and every precondition P and postcondition Q.

It is customary to prove completeness for Hoare logics under two additional assumptions. The first assumption is that every valid formula of the assertion language is an axiom of the Hoare logic [2]. This assumptions enables the completeness proof to focus on the strength of the Hoare rules instead of the underlying proof system for the assertion language. The second assumption concerns the expressiveness of the assertion language; one only proves completeness for interpretations of the assertion language that allow one to express the strongest postcondition or the weakest precondition of every statement in the assertion language [5, 2]. This notion of completeness is known as *relative* completeness.

We believe that relative completeness is not the right completeness criterion for program logics that are based on behavioral subtyping; it is too stringent for such logics. Recall that behavioral subtyping forces subtypes to behave in the same way as their supertype. However, the standard relative completeness notion does not restrain subtypes at all. This leads to a problem, which we will illustrate using an example.

Consider the following two classes that each model a clock.

```
class Clock extends Object              class EnhClock extends Clock
    refines Object {                        refines Clock {
  int maxHour;
  void  init() {                          void  init() {
    maxHour := 12; }                        maxHour := 24; }
}                                       }
```

The first class *Clock* models a simple clock that represents, for example, 23:00 (11:00 PM) by 11:00. Its field *maxHour*, which stores the maximum value of the (omitted) hour field, is initialized to 12. Its more enhanced subclass *EnhClock* initializes *maxHour* to 24 by overriding the *init*-method that it inherits from *Clock*. The other fields and methods of the classes do not play a role in our example and are therefore omitted. We assume that *Object* is the name of the root class.

Now consider the following Hoare triple.

$$\{\ \mathsf{true}\ \}\ c := \mathsf{new}\ \mathit{Clock}();\ \ c.\mathit{init}();\ \ \{c.\mathit{maxHour} = 12\} \qquad (1)$$

This Hoare triple is valid in every possible program that includes class *Clock* as defined above. The dynamic type of the new clock is *Clock*, and the subsequent call to the *init*-method will therefore be bound to the implementation of the method in that class.

The Hoare triple (1) can be derived in a Hoare-like logic if we annotate the *init*-method in class *Clock* with the specification

$$\{\mathsf{true}\}\ \mathit{init}@\mathit{Clock}\ \{\mathsf{this}.\mathit{maxHour} = 12\}\ .$$

However, we must now show that the implementation in class *EnhClock* that overrides the original *init*-method also satisfies this specification, which is clearly not the case.

There is a way out that is often used in this kind of situations. One can change the method specification such that it does not confine overriding methods by restricting the specification to receivers of a particular type. For this purpose, we introduce a function type that yields the dynamic type of an object. The specification of the *init*-method then becomes

$$\{\mathsf{true}\}\ \mathit{init}@\mathit{Clock}\ \{\mathsf{type}(\mathsf{this}) = \mathit{Clock} \Rightarrow \mathsf{this}.\mathit{maxHour} = 12\}\ . \qquad (2)$$

This specification also holds for the implementation in class *EnhClock* because the receiver has static type *EnhClock* in that class. Hence this cannot denote a *Clock*-object. Moreover, the specification still suffices to prove (1).

However, we now run into another problem. Imagine that we add a class *FancyClock* (another subclass of class *Clock*) to our program that does *not* override method *init*. Then we should be able to prove that

$$\{\ \mathsf{true}\ \}\ c := \mathsf{new}\ \mathit{FancyClock}();\ \ c.\mathit{init}();\ \ \{c.\mathit{maxHour} = 12\}$$

is a valid Hoare triple. But (2) is too weak to prove this specification. Naturally, we can change (2) such that it also supports the new class *FancyClock*. But then we can repeat our counterexample with another class. The proposed way out only seems to solve the problem for closed programs in which all subclasses are know beforehand. This is not satisfactory because the main reason for using behavioral subtyping is that it allows one to reason about *open* programs.

4 Behavioral Completeness

The example in the previous section shows that the standard completeness notion is too stringent for program logics that are based on behavioral subtyping. The problem is that the standard completeness notion does not restrict the set of programs that must be considered. As a consequence, program logics must also be able to prove properties of programs in which subtypes occur that do not reveal the same behavior as their supertype, which seems to be undesirable.

For this reason, we propose a new notion of completeness that restricts the set of programs that must be considered. We will require that every program satisfies the behavioral subtyping principle. A program satisfies the *behavioral subtyping principle* if each class C in the program that is declared to refine another class D is indeed a behavioral subtype of class D. Moreover, for an open program we require that the program is only extended with classes that preserve the behavioral subtyping principle.

We propose to call this novel completeness notion *behavioral* (subtype) *completeness*. It is defined as follows.

Definition 1 (behavioral completeness). *A program logic is behavioral complete if and only if* $\models \{P\}\ S\ \{Q\}$ *implies* $\vdash \{P\}\ S\ \{Q\}$ *for every correctness formula* $\{P\}\ S\ \{Q\}$ *in the context of a program that satisfies the behavioral subtyping principle.*

Note that every Hoare logic operates in the context of some particular program, a fact which is usually not directly reflected in the Hoare triples for brevity. We omitted the two additional assumptions of relative completeness (see Section 3) in our definition of behavioral completeness. However, the definition above should be read as an extension of the definition of relative completeness. In other words, the two assumptions of relative completeness are also allowed in order to prove behavioral completeness of a proof system. Thus behavioral completeness becomes a truly weaker property than relative completeness.

The problem that we signalled in the previous section for relative completeness does not apply to behavioral completeness. The class *EnhClock* is not a behavioral subtype of class *Clock* because its implementation of the *init*-method does not satisfy the valid specification

$$\{\ \mathsf{true}\ \}\ init@Clock\ \{\mathsf{this}.maxHour = 12\}$$

of the implementation that it overrides. The class can nevertheless be allowed in a program if it is only declared as a subclass of *Clock*, and not as a subtype of *Clock*. In other words, it would be allowed if its first line would read

$$\mathsf{class}\ EnhClock\ \mathsf{extends}\ Clock\ \mathsf{refines}\ Object\ \ .$$

The standard approach to proving relative completeness of a Hoare logic, which has been proposed by Gorelick [9], annotates every method with its Most General (correctness) Formula, and subsequently shows that each valid correctness formula of a statement can then be derived (see also [6]). This approach cannot be followed if a program calls methods from predefined modules or packages. More precisely, it does not support *reuse* of the supplied method specifications. We will investigate a completeness notion that supports proof reuse in the following section.

5 Behavioral Modular Completeness

Object-oriented programs usually heavily depend on functionality that is provided by imported classes. Ideally, the methods in these classes are already annotated with specifications, which then can be reused. In this scenario, one can treat the provided methods as black boxes, and reason about calls to these methods on the basis of their specifications.

Viewing imported methods as black boxes has consequences for the requirements that subtypes should meet. The definition of the behavioral subtyping principle that we gave in the previous section does not treat a method as a black box. It requires that an overriding method satisfies every valid specification of the method that it overrides. An overriding method in a subtype can only meet this criterion if it has knowledge about the (hidden) implementation of the method.

We will therefore adopt a different criterion for subtypes in this section. Essentially, we will require that each method that overrides a method in a supertype has a specification that refines the specification of the overridden method. We say that a particular *specification spec refines the specification spec′ of another method* if every statement that satisfies *spec* also satisfies *spec′*. (We assume here that a method has only one specification.) A *class C* is said to *refine a supertype D* if every method of C that overrides a method of D has a specification that refines the specification of the overridden method. Finally, we say that *a program satisfies the specification refinement principle* if every class in the program refines its supertypes.

The choice to view imported methods as black boxes also has consequences for the completeness of a program logic. The provided specifications of the imported methods may be weaker than the strongest possible specifications that would be required to prove all valid specifications of calls to those methods. Hence it may be impossible to prove behavioral completeness in these circumstances.

This latter phenomenon has already been observed by Zwiers. He responded to this problem by introducing the notion of *modular completeness* [16]. We propose the following variant (dubbed *behavioral* modular completeness) of modular completeness for proof systems that are based on behavioral subtyping.

Definition 2 (behavioral modular completeness). *A formal proof system is behavioral modular complete if, for every Hoare triple $\{P\}\ S\ \{Q\}$, and for all method specifications $\{P_i\}\ m_i@C_i\ \{Q_i\}$, where $i \in \{1 \ldots n\}$, we have that*

$$\{P_1\}\ m_1@C_1\ \{Q_1\}, \ldots, \{P_n\}\ m_n@C_n\ \{Q_n\} \models \{P\}\ S\ \{Q\}$$

implies

$$\{P_1\}\ m_1@C_1\ \{Q_1\}, \ldots, \{P_n\}\ m_n@C_n\{Q_n\} \vdash \{P\}\ S\ \{Q\}\ ,$$

in the context of a program that satisfies the specification refinement principle.

There are several elements of this definition that deserve some clarification. We will start with the last formula, which says that we can derive $\{P\}\ S\ \{Q\}$ using the assumptions $\{P_1\}\ m_1@C_1\ \{Q_1\}, \ldots, \{P_n\}\ m_n@C_n\{Q_n\}$. We will assume that $m_1@C_1, \ldots, m_n@C_n$ are mutually distinct to maintain our earlier assumption that methods only have one specification.

The method enumeration $m_1@C_1, \ldots, m_n@C_n$ is the set of methods that are in scope in the context of the statement S. It contains at least the methods that statically match the method calls that occur in S. That is, if S contains a method call $e.m(\ldots)$, where $[e] = C$, then we assume that the method $m@C'$ of class C occurs in the enumeration $m_1@C_1, \ldots, m_n@C_n$.

An interesting phenomenon arises if the enumeration contains methods that override methods that statically match a particular call in S. The specification of such an overriding method may provide additional information because the behavioral subtyping principle forces it to have a specification that refines the specification of the overridden method. In other words, it may have a stronger specification. If the receiver of the call belongs to the domain of the class of the overriding method, then we know that the implementation that is bound to the call satisfies the specification of the overriding method. Hence we may reason about the call using the stronger specification of the overriding method.

We will use this observation to assign a meaning to the antecedent

$$\{P_1\}\ m_1@C_1\ \{Q_1\}, \ldots, \{P_n\}\ m_n@C_n\ \{Q_n\} \models \{P\}\ S\ \{Q\} \tag{3}$$

of the implication in the definition. This equation roughly says that $\{P\}\ S\ \{Q\}$ is a valid Hoare triple if the Hoare triples $\{P_1\}m_1@C_1\{Q_1\}, \ldots, \{P_n\}m_n@C_n\{Q_n\}$ are also valid.

One can formalize this notion by defining an (operational semantics) relation $\langle S, \sigma\rangle \xrightarrow{SPEC} \sigma'$ between configurations $\langle S, \sigma\rangle$, which consist of a statement S and an initial state σ, and final states σ'. The parameter *SPEC* of the relation denotes a finite set of method specifications

$$\{P_1\}\ m_1@C_1\ \{Q_1\}, \ldots, \{P_n\}\ m_n@C_n\ \{Q_n\}\ .$$

By $\langle S, \sigma\rangle \xrightarrow{SPEC} \sigma'$ we denote that a computation of S that starts in state σ may terminate in state σ' if the methods that are called by S satisfy their specifications in *SPEC*.

We model the state σ of an object-oriented program as a pair (s, h) that consists of a stack s and a heap h. The stack s is a function that assigns values to simple variables. A heap h specifies the set of objects that exists in a state, and the values of their fields (see [15] for more details).

The relation is defined as usual by a set of rules and axioms (cf., e.g., [14]). The only non-standard rule is the rule for method calls. It is the only rule that takes the set *SPEC* into account. It specifies when

$$\langle e_0.m(e_1, \ldots, e_k), (s, h)\rangle \xrightarrow{SPEC} (s, h') \tag{4}$$

holds. The validity of (4) is determined by the specifications of the methods in *SPEC*; methods specifications that are not in scope and specifications of future methods are ignored. More specifically, if C is the dynamic type of the object denoted by e_0 in the state (s, h), then we use the specification of the most specific implementation of m in *SPEC* that is inherited by C to determine whether (4) holds (a method is *more specific* than another method if it overrides that method). Let $\{P\}\ m@C\ \{Q\}$ be this specification. Then (4) is true if the following conditions hold.

– The precondition P holds in the state (s'', h), where s'' is obtained from s by assigning the value of e_0 in (s, h) to `this`, and the values of $e_1, \ldots, e_k$ in (s, h) to $p_1, \ldots, p_k$.

– The heap h' is an extension of h, i.e., every object that exists in h also exists in h' (we do not take object deallocation into account).
– The postcondition Q holds in the state (s, h').

With the defined relation we can give a formal meaning to expressions of the form of Eq. (3). We have $\mathit{SPEC} \models \{P\}\ S\ \{Q\}$ if for every state σ that satisfies P, and every final state σ' such that $\langle S, \sigma\rangle \overset{\mathit{SPEC}}{\longrightarrow} \sigma'$, we have that σ' satisfies Q.

By defining the $\overset{\mathit{SPEC}}{\longrightarrow}$-relation in terms of the most specific available method specification we get a larger set of valid specifications $\{P\}\ S\ \{Q\}$, and consequently a stronger completeness criterion. Alternatively, we could have used the specification of the method m that belongs to the methods of the static type of the receiver expression e_0. The most specific method is a method that is defined in a subtype of $[e_0]$. Hence it may have a stronger specification.

Example 1. Suppose that we have a class *Point* with a subtype *ThreeDPoint*. Class *Point* has two integer fields x and y, and class *ThreeDPoint* has an additional z field. Suppose that each of the two classes implements a *reset* method. We will assume that these two methods have the following specifications.

$$\{\mathsf{true}\}\ \mathit{reset}@\mathit{Point}\ \{\mathsf{this}.x = 0 \land \mathsf{this}.y = 0\} \tag{5}$$

$$\{\mathsf{true}\}\ \mathit{reset}@\mathit{ThreeDPoint}\ \{\mathsf{this}.x = 0 \land \mathsf{this}.y = 0 \land \mathsf{this}.z = 0\} \tag{6}$$

Note that the specification of *reset* in class *ThreeDPoint* refines the specification of *reset* in class *Point*. Next, consider the following Hoare triple

$$\{p\ \mathsf{instanceof}\ \mathit{ThreeDPoint}\}\ p.\mathit{reset}()\ \{((\mathit{ThreeDClock})p).z = 0\}\ , \tag{7}$$

where p is a local variable of type *Point*. Suppose that the call $p.\mathit{reset}()$ occurs in a context in which both classes are in scope. Then (7) is a valid specification because in each state that satisfies the precondition of (7) we have that the dynamic type of the receiver is an instance of (a subclass of) *ThreeDPoint*. Hence its most specific implementation of *reset* is the one in class *ThreeDPoint*. The Hoare triple would not be valid if we would have based our semantical relation on the specification of the method that corresponds to the static type of p because that would be the specification in class *Point* which says nothing about the value of the z field of its receiver.

6 Conclusions

This paper reveals several interesting aspects of the completeness of logics that are based on behavioral subtyping. We have argued that relative completeness is not the right criterion for logics that are based on behavioral subtyping, and subsequently proposed two completeness criteria that are more suitable for these logics. The outlined criteria can be used to clarify the (possibly intentional) incompleteness of existing logics. We also hope that they foster research that may result in logics that satisfy one of the completeness criteria, or even both.

References

1. P. America. Designing an object-oriented programming language with behavioral subtyping. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, volume 489 of *LNCS*, pages 60–90. Springer, 1991.
2. K. R. Apt. Ten Years of Hoare's Logic: A Survey - Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.
3. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
4. D. R. Cok and J. R. Kiniry. ESC/Java2: Unitying ESC/Java and JML. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *LNCS*, pages 108–128. Springer, 2005.
5. S. A. Cook. Soundness and completeness of an axiom system for program verification. *Siam Journal of Computing*, 7(1):70–90, February 1978.
6. F. S. de Boer and C. Pierik. How to Cook a complete Hoare logic for your pet OO language. In *Formal Methods for Components and Objects (FMCO 2003)*, volume 3188 of *LNCS*, pages 111–133. Springer, 2004.
7. K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering (ICSE '96)*, pages 258–267. IEEE Computer Society Press, 1996.
8. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of Programming Language Design and Implementation (PLDI 2002)*, pages 234–245. ACM Press, 2002.
9. G. Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report 75, Dep. Computer Science, Univ. Toronto, 1975.
10. G. T. Leavens and K. K. Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.
11. G. T. Leavens and W. E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, 1995.
12. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Programming Language Design and Implementation (PLDI 2002)*, pages 246–257. ACM Press, 2002.
13. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
14. D. A. Naumann. Observational purity and encapsulation. In *Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442 of *LNCS*, pages 190–204. Springer, 2005.
15. C. Pierik and F. S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, volume 2884 of *LNCS*, pages 64–78, 2003.
16. J. Zwiers. *Compositionality, Concurrency, and Partial Correctness*, volume 321 of *LNCS*. Springer-Verlag, 1987.