6

Concepts of Behavioral Subtyping and a Sketch of their Extension to Component-Based Systems

Gary T. Leavens
Department of Computer Science, 226 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
leavens@cs.iastate.edu

Krishna Kishore Dhara
Bell Laboratories, Lucent Technologies
2f-247, 600-700 Mountain Ave
Murray Hill, NJ 07974-0636 USA
dhara@lucent.com

Abstract

Object-oriented systems are able to treat objects indirectly by message passing. This allows them to manipulate objects without knowing their exact runtime type. Behavioral subtyping helps one reason in a modular fashion about such programs. That is, one can reason based on the static types of expressions in a program, provided that static types are upper bounds of the runtime types in a subtyping preorder, and that subtypes satisfy the conditions of behavioral subtyping. We survey various notions of behavioral subtyping proposed in the literature for object-oriented programming. We also sketch a notion of behavioral subtyping for objects in component-based systems, where reasoning about the events that a component can raise is important.

6.1 Introduction

Component-based systems require a renewed emphasis on specification and verification, because if one is to build a computer system based on components built by others, then one must know what each component is supposed to do and trust it to carry out that task. Similarly, the builder of a component needs to know what behavior its users depend on, so that improvements in algorithms and data structures can be made.

A specification of a component can meet both these needs, since it acts as a contract between builders and their clients [LG86, Mey92]. The builders are obligated to make the component behave as specified, but gain the opportunity to use any data structures and algorithms that satisfy the contract. A client can only use the component through the specified interface given by the contract; in particular the client is prohibited from using hidden features. In return, the client gains the ability to treat the component abstractly, as a black box that behaves as specified.

6.1.1 Background

Traditionally, software has been specified using pre- and post-conditions [Hoa69, Dij76, Hes92]. As is well-known, a procedure specification given in this style consists of two predicates. The *precondition* describes the states in which the procedure can be invoked; if procedures are modeled as relations between states (pre-state and post-states), then the precondition is the characteristic predicate of this relation's domain. The *postcondition* describes the transformation of pre-states into acceptable post-states; it is the characteristic predicate of the relation itself.

Abstract data types (ADTs) can also be specified using such specifications for their operations; these specifications are written using a mathematical abstraction of the values of objects of the type, called *abstract values* [FJ92, GHG$^+$93, Jon90, LG86, OSWZ94]. To prove an implementation of such a specification is correct, one must be able to find an abstraction relation that relates the values of the objects used in the implementation to the abstract values, in such a way that the relationship is preserved by the operations, and is the identity on more fundamental types (like the integers) [Hoa72, LP97, Nip86, Sch90, SWO97].

For components in the sense of Microsoft COM or Java Beans, specification techniques are much less clear. Key features of components that are distinct from OO systems and that affect specification and verification are the following [Szy98, DW99].

- A component may provide more than one interface to its clients. For instance, it will typically provide an interface for other components (listeners) to register for the events that it may raise. However, each such interface can be specified separately.
- A component may not be self-contained, but may have some requirements on the context in which it must be used. However, one can treat these dependencies as extra parameters, as, for example, is done in OBJ [FGJM85, Gog84] and the RESOLVE family of specification languages [SWO97].
- A component will raise events (i.e., invoke callbacks) during execution of its operations, for example when its instances experience state changes. Traditional specification languages ignore such higher-order behavior, although the refinement calculus [Bac88, Bv92, BvW98, Mor94, MV94, Mor87, Woo91] does provide a paradigm for specifying when such events are raised by using model (or abstract) programs. (But it is only recently that it has been applied in this area [BS97, Mik98a, MSL99].)

  For example, an editor component might have a context dependency on a spell checker, and would raise events when the text is changed.

6.1.2 Specifications for Components

As we described above, the higher-order behavior of callbacks, is a critical issue that distinguishes the specification of components from the specification of ADTs. By a *callback*, we mean a method that is invoked to handle some event; typically this method is in an object that is only registered as interested in the event at run-time.

Szyperski illustrates the pitfalls of callbacks with examples such as directories and a text models and views [Szy98, Section 5.5]. His directory example illustrates the complicating factor of callbacks sharply. Consider a `Directory` component with operations in its interface to add and remove files: `addEntry` and `removeEntry`. Unlike a simple, first-order OO ADT, these operations also raise events to notify listeners of changes in the directory. The listeners satisfy an interface named `DirObserver`. The class `Directory` also provides operations to register and unregister listeners for such events, which it inherits from its ancestor `DirObserverKeeper`. The immediate superclass of `Directory` is a class `RODirectory`, which is a subclass of `DirObserverKeeper`. `RODirectory` supplies methods `equals` and `thisFile` to observe directories.

However, if the specification for `Directory` does not take the callbacks into account, then there is no way to guarantee the postconditions of `addEntry` and `removeEntry`, since a callback can undo the work of either of them. In Szypreski's example, invoking `addEntry` with a file named "`Untitled`" breaks the contract because of the behavior of a callback that removes files named "`Untitled`". However, the caller (client) thinks that `addEntry` is broken when in fact the behavior of the callback is the one that caused the anomaly.

Szyperski gives a specification that prevents this problem by having a test function `inNotifier` that returns true when "a notifier call is in progress" [Szy98, page 56]. He adds preconditions to `addEntry` and `removeEntry` that require that `inNotifier` returns false; so that changes to directories can only occur when no notification is in progress. This is fine, but leaves one to wonder how the details can be formalized.

One way to formalize the specification of this example is to use both specification-only variables (also called ghost or model variables, see, for example Leino's work [Lei95]) and model programs (as in the refinement calculus). We present such a specification for the four types written in JML [LBR99] below.

Figure 6.1 specifies the type `DirObserverKeeper`. It is itself a fairly simple ADT.

Some notes on JML may be helpful. JML is a behavioral interface specification language. It specifies the interface of a Java module using Java syntax, and adds annotations to specify behavior. In JML, Java comments that start with an at-sign (`@`) mark annotations; JML treats the body of such comments as part of the specification.

As in Larch [GHG+93], specifications of behavior are written in terms of abstract values, which are given as the values of model variables in JML. In JML specification-only declarations use the keyword `model:`. The keyword `instance:`

```
import edu.iastate.cs.jml.models.*;

public interface DirObserverKeeper extends JMLType {
    //@ public model: instance: boolean in_notifier
    //@                          initially in_notifier == false;
    //@ public model: instance: JMLObjectSet listeners
    //@                          initially listeners != null
    //@                            && listeners.equals(new JMLObjectSet());
    //@ public invariant: listeners != null;
  public boolean inNotifier();
    //@ normal_behavior:
    //@   ensures: \result == in_notifier;

  public void register(DirObserver o);
    //@ normal_behavior:
    //@   requires: o != null;
    //@   modifiable: listeners;
    //@   ensures: listeners.equals(\old(listeners).insert(o));

  public void unregister(DirObserver o);
    //@ normal_behavior:
    //@   requires: o != null;
    //@   modifiable: listeners;
    //@   ensures: listeners.equals(\old(listeners).remove(o));
}
```

Fig. 6.1. A JML specification of a `DirObserverKeeper` interface

says that a field declaration, which in an interface would normally be `static`, is instead to be considered as an instance variable in each class that implements the interface. For example, in Figure 6.1, `in_notifier` and `listeners` are both *model variables*. The `initially` clauses give possible starting values for these model variables. The abstract values of these model variables are described, to the user, as either built-in Java primitive types (like `boolean`) or as a Java class with immutable objects. JML calls such classes *pure*; they are used to encapsulate the mathematical description of abstract values. An example of such a class is `JMLObjectSet`, which is found in the package `edu.iastate.cs.jml.models`. Such classes allow JML specifications to use Java expression syntax for invoking their operations, without compromising mathematical rigor. The operations (Java methods) are specified with `normal_behavior:` clauses, which give the usual pre- and postcondition style specification over the model variables. Preconditions start with `requires:`, and postconditions with `ensures:`. The `modifiable:` clauses in the last two method specifications say that only the model variable `listeners`, and variables declared to depend on it in an implementation [LBR99, Lei95], may have their value changed by invocation of one of these methods.

Figure 6.2 specifies the interface of directory observers, which are callback objects.

```
import Directory;

public interface DirObserver {
  void addNotification(Directory o, String n);
  //@ normal_behavior:
  //@   requires: o != null && o.in_notifier && n != null && n != "";
  //@   modifiable: \everything;
  //@   ensures: o.equals(\old(o));

  void removeNotification(Directory o, String n);
  //@ normal_behavior:
  //@   requires: o != null && o.in_notifier && n != null && n != "";
  //@   modifiable: \everything;
  //@   ensures: o.equals(\old(o));
}
```

Fig. 6.2. A JML specification of a `DirObserver` interface.

The callbacks will be made when the model variable `in_notifier` of the directory that is being changed is true. The callbacks are permitted to do anything at all, except that they must terminate normally, and cannot modify the directory giving notice. The `\old(o)` in the postcondition represents the pre-state value of `o`.

Figure 6.3 specifies the interface `RODirectory`, for read-only directories. This interface extends `DirObserverKeeper`, and so inherits all of its specifications [DL96, LBR99]. It adds a model variable `entries`, which is a finite map from strings to file objects. The two `invariant:` clauses specify these types. The `thisFile` operation is specified using case analysis; the first `requires:` clause applies globally, but only one of the two specification cases that follow it will apply, depending on whether the name given is defined in the map `entries`. The `equals` operation is also respecified here.

Figure 6.4 gives the specification of the interface `Directory`, which adds `AddEntry` and `RemoveEntry` methods to its superclass `RODirectory`. Both specifications are similar, so let us consider the specification of `AddEntry`. It is given by a model program (hence it starts with `model_program:`). The syntax of a model program is that of a Java *block* (statements surrounded by curly braces). As in the refinement calculus, the meaning is that a correct implementation must refine the model program. This model program in `AddEntry` contains two statements. The first is a `normal_behavior:` statement, which is followed by a `for`-loop. The normal behavior statement ends at the semicolon (`;`) following `ensures:`. It is a specification of what, in a refinement, some concrete code must accomplish. It says that `entries` is modified to add the given association. The `for`-loop is used to say, abstractly, how notifications are done. From this one can tell that when `addNotification` is called, the model variable `in_notifier` is true, and the association has already been added to the directory.

```
//@ model: import edu.iastate.cs.jml.models.*;

public interface RODirectory extends DirObserverKeeper {
    //@ public model: JMLValueToObjectMap entries
    //@        initially entries != null
    //@                  && entries.equals(new JMLValueToObjectMap());
    //@ public invariant: entries != null && \forall (JMLType o)
    //@         [entries.isDefinedAt(o) ==> o instanceof JMLString];
    //@ public invariant: \forall (JMLString s)
    //@         [entries.isDefinedAt(s)
    //@           ==> entries.apply(s) instanceof Files.File];

  public Files.File thisFile(String n);
    //@ normal_behavior:
    //@   requires: n != null && n != "";
    //@   {
    //@      requires: entries.isDefinedAt(new JMLString(n));
    //@      ensures: \result.equals( (Files.File)
    //@                              entries.apply(new JMLString(n)));
    //@    also:
    //@      requires: !entries.isDefinedAt(new JMLString(n));
    //@      ensures: \result == null;
    //@   }

  public /*@ pure: @*/ boolean equals(Object oth);
    //@ normal_behavior:
    //@     requires: !(oth instanceof RODirectory);
    //@     ensures: \result == false;
    //@   also:
    //@     requires: oth instanceof RODirectory;
    //@     ensures: \result ==
    //@        (   entries.equals(((RODirectory)oth).entries)
    //@         && listeners.equals(((RODirectory)oth).listeners));
}
```

Fig. 6.3. A specification of a `RODirectory` interface.

The technical "tricks" used in this specification were model (ghost) variables and model programs. By using these features, one can show both what callbacks can do and exactly the state in which they are called. The utility of model programs in this setting was first made known to us by Büchi and Sekerinski [BS97]. For us, model programs seem more practical than other ways of specifying callbacks and higher-order procedures [EHL+94, EHMO91, Gog84]. Recently, other work on the refinement calculus has provided a more thorough treatment of this subject, including a modular reasoning technique that has been proved sound [MSL99].

```
//@ model: import edu.iastate.cs.jml.models.*;

public interface Directory extends RODirectory {

  public void addEntry(String n, Files.File f);
  //@ model_program: {
  //@   normal_behavior:
  //@     requires: !in_notifier && n != null && n != "" && f != null;
  //@     modifiable: entries;
  //@     ensures: entries != null
  //@        && entries.equals(\old(entries.extend(new JMLString(n), f)));
  //@   for (JMLObjectSetEnumerator e
  //@                          = new JMLObjectSetEnumerator(listeners);
  //@        e.hasMoreElements(); ) {
  //@     in_notifier = true;
  //@     ((DirObserver)e.nextElement()).addNotification(this, n);
  //@     in_notifier = false;
  //@   }
  //@ }

  public void removeEntry(String n);
  //@ model_program: {
  //@   normal_behavior:
  //@     requires: !in_notifier && n != null && n != "";
  //@     modifiable: entries;
  //@     ensures: entries != null
  //@        && entries.equals(\old(entries.remove(new JMLString(n))));
  //@   for (JMLObjectSetEnumerator e
  //@                          = new JMLObjectSetEnumerator(listeners);
  //@        e.hasMoreElements(); ) {
  //@     in_notifier = true;
  //@     ((DirObserver)e.nextElement()).removeNotification(this, n);
  //@     in_notifier = false;
  //@   }
  //@ }
}
```

Fig. 6.4. A specification of a `Directory` interface.

6.1.3 Modular Reasoning

An important concern in both object-oriented and component-based programming is how to reason about extensions of programs. For example, suppose one has a method m that takes a `RODirectory` as an argument. If the implementation of m is correct with respect to its specification, then it should work correctly for `Directory` arguments. This is the notion of *modular reasoning*, which can be seen as a criteria for goodness of verification techniques [LW90, LW95, Lei95]. The basic idea for modular reasoning about OO programs is to:

- Assign to each expression in a program a static type that is an upper bound on the dynamic type of the expression's value. (That is, if the static type is T, then the value must have a dynamic type that is a subtype of T.)
- Reason about client code using the static types of expressions, as in standard reasoning about programs with ADTs.
- Prove that each subtype used in the program is a behavioral subtype of its supertypes [Ame87, AvdL90, Ame91, Dha97, LW93b, LW94, Utt92, UR92]. In simplest terms, this means that the subtype objects obey the specification of their supertype objects [DL96].

The advantage of modular reasoning is that unchanged methods do not have to be respecified or reverified when new behavioral subtypes are added.

6.1.4 Outline

The rest of this chapter is organized as follows. In Section 6.2 we consider the relationship between subtypes and behavioral subtypes. In Section 6.3 we survey the literature on behavioral subtyping in OO systems. We then discuss in Section 6.4 some ideas about subtyping for component-based systems. Finally, we offer some conclusions.

6.2 Subtyping and Behavioral Subtyping

In this section we define some important terms and make several distinctions among superficially-related concepts in OO languages that are important in understanding behavioral subtyping and how they differs from less OO concepts. In particular we distinguish meta-types and object types, and refinement and behavioral subtyping.

6.2.1 Classes, Types, and Specifications

A *class* is a program module that describes a set of potential *instances* or *objects*. In many languages, such as Java and Smalltalk, a class also describes a *class object*, which can be sent messages to create instances (in Smalltalk), and which also holds information common to all instances (such as the code for methods, the class name, etc.). One can also make a distinction between instance methods and class methods; *instance methods* can be sent to an instance, while *class methods* are sent to class objects. We will use the term "class method" to refer also to the static methods and constructors of languages like C++ and Java.

A *type* is a static attribute of some phrase in a programming language. For example, numeric literals have a type such as `int`. In OO languages, both class objects and instances have types. The type of an instance is derived from the class declaration, and a structural rendering of such a type only involves the instance methods. Such a type corresponds to a Java interface, since it describes a protocol for manipulating objects; hence it is called an *object type*. The extension of an object type is thus a set of objects with a common protocol [GHJV95]. All the instances of a given object type can thus be sent the same set of messages (method calls with arguments) without generating a type error.

By contrast a class type or *meta-type* describes the protocol of class objects. A structural rendering of such a type involves the types of class methods and the object types of the instances that the class can create.

Types can be viewed as degenerate specifications, since they give information about the syntax of methods (their names, and types of arguments, etc.), but do not (usually) involve behavior. By contrast, a *behavioral specification* describes both syntax and semantics, as seen in the preceding figures.

A behavioral specification of a meta-type (i.e., of a set of classes) involves both the specification of how objects are created (constructors in C++ and Java), class methods, and a specification of how instances behave in response to instance methods. By contrast, a behavioral specification of an object type (a Java interface) does not involve constructors or other class methods.

6.2.2 Refinement

Refinement is an important relationship on meta-types. It is a stronger relationship than behavioral subtyping, which relates object types.

Refinement is a relationship between behavioral specifications that is useful in developing programs from specifications [Mor94, Woo91]. The basic idea is that a *refinement*, C, of a specification, A, is a specification that is stronger than A in the sense that every correct implementation of C is also a correct implementation of A; thus, C will have no more correct implementations than A. Another way of thinking about a refinement is that the set of allowed behaviors of the refinement is a subset of the behaviors allowed by the original specification.

Refinement can also be extended to a relationship between implementations and specifications and between implementations. If one thinks of each implementation module as having a specification that describes its exact behavior, or if one uses a programming language as an (operational) specification language, then this idea, which is crucial to the refinement calculus, falls out. In this sense, one meaning of "a refinement" is "a correct implementation." Thus, we will say just "a refinement" for the longer phrase "an implementation of a refined specification" below.

For example, given a procedure specification, one can reason about the correctness of client code using its specification, without knowing anything about its implementation. Many different implementations may be linked into a program without changing the soundness of such reasoning, if each such implementation is a refinement of the specification used in reasoning. If one takes a total-correctness specification for a procedure g with precondition R_A and postcondition E_A, then a refinement must:

- have the same syntax (the same name, number of arguments, argument types, return type, and exception result types),
- a precondition R_C such that $R_A \Rightarrow R_C$, and
- a postcondition E_C such that the following holds.

$$(R_C \Rightarrow E_C) \Rightarrow (R_A \Rightarrow E_A) \tag{6.1}$$

See the paper by Cheng and Chen in this volume for further discussion of formula (6.1). This formula is weaker than the usual one for postconditions, which is that $E_C \Rightarrow E_A$ [Mor90]. The usual formula is both simpler and works in most practical cases. Note also that termination is implicitly required by both specifications, and so there is no explicit proof obligation to show termination.

For specifications given in terms of model programs, the techniques of the refinement calculus would be used instead of the pre- and postcondition rule described above.

For abstract data types, refinement means again that each implementation of refinement is an implementation of the original specification [Win83, GM94]. Such data type refinement can be mediated by a change in the way data is modeled [GM94, MG90, Mor94, Mor89]. One can use an abstraction function [Hoa72] or relation [LP97, Nip86, Sch90, SWO97] to translate between logical assertions in the theory of one abstract model and another. For example, suppose the specification C is a refinement of A, and C is stated using a theory T_C, which we assume includes the theory used to state the specification of A. Suppose that $r_{C\rightarrow A}$ is a relation between the models of C and A, so that $r_{C\rightarrow A}(c', a')$ holds when c' is related to a'. In the following we use the notations $x\,\hat{}$, $x\,'$, and x° to denote the pre-state, post-state, and arbitrary public state values of x (respectively). Then it must be that (again for total-correctness specifications):

- C and A have the same interface (the same name, class and instance methods with the same number of arguments, argument types, etc.),
- using the theory of the specification of C, T_C, C's invariant implies A's

$$\begin{aligned} T_C \;\;\vdash\;\; & \forall \mathit{self}: C \,.\, \exists x : A \,. \\ & r_{C\rightarrow A}(\mathit{self}^{\circ}, x^{\circ}) \\ & \wedge (\mathit{invariant}_C(\mathit{self}^{\circ}) \Rightarrow \mathit{invariant}_A(x^{\circ})) \end{aligned} \tag{6.2}$$

- C's history constraint† must imply A's:

$$\begin{aligned} T_C \;\;\vdash\;\; & \forall \mathit{self}: C \,.\, \exists x : A \,. \\ & r_{C\rightarrow A}(\mathit{self}\,\hat{}, x\,\hat{}) \wedge r_{C\rightarrow A}(\mathit{self}', x\,') \\ & \wedge (\mathit{constraint}_C(\mathit{self}\,\hat{}, \mathit{self}') \Rightarrow \mathit{constraint}_A(x\,\hat{}, x\,')) \end{aligned} \tag{6.3}$$

† A history constraint is a monotonic relation on pairs of states; it relates an earlier state to a later state [LW93b, LW94]. History constraints are useful in abbreviating specifications, and have implications for behavioral subtyping that are discussed below. In JML history contraints are syntatically stated as if they related a pre-state and a post-state, although the semantics is more general.

- for each instance method, g, the specification of g in C must refine that of g in A via $r_{C\to A}$. For pre- and postcondition specifications, this means that:
  - g's precondition in A, pre^g_A, must imply the corresponding precondition in C:

$$T_C \;\vdash\; \forall self : C \,.\, \exists x : A \,.\; r_{C\to A}(self\hat{}, x\,\hat{}) \wedge (pre^g_A(x\,\hat{}) \Rightarrow pre^g_C(self\hat{})) \tag{6.4}$$

  - g's specification in C must be such that the following holds:

$$T_C \;\vdash\; \forall self : C \,.\, \exists x : A \,.\; r_{C\to A}(self\hat{}, x\,\hat{}) \wedge r_{C\to A}(self', x\,') \wedge (((pre^g_C(self\hat{}) \Rightarrow post^g_C(self\hat{}, self')) \Rightarrow (pre^g_A(x\,\hat{}) \Rightarrow post^g_A(x\,\hat{}, x\,')))) \tag{6.5}$$

- each class method, f, in A is refined by the class method f in C via $r_{C\to A}$.

Besides dealing with model programs, the refinement calculus is a way of systematically deriving refinements [Bac88, Bv92, BvW98, Mor94, MV94, Mor87, Woo91]. Each such derivation is a small step, and is guaranteed to be correct. The calculus uses a wide-spectrum language in which programs are enriched by (nondeterministic) specification statements. In this way one may start the refinement process with a behavioral specification which consists of only a specification statement, and by making several refinement steps, arrive at completely executable code.

6.2.3 Subclasses, Subtypes, and Behavioral Subtypes

Refinement, as defined above, does not capture one key feature of OO programming: the use of message passing to achieve subtype polymorphism. One way to view this distinction is that refinement for abstract data types makes no distinction between meta types and object types. Behavioral subtyping is essentially refinement of object types, whereas in common terminology, refinement of types refers to meta-types.

However, let us step back for a moment, and discuss not the relation of refinement and behavioral subtyping, but the relationships of subclassing, subtyping, and behavioral subtyping.

Since classes, types, and behavioral specifications are, in our terminology, different kinds of things, it follows that subclassing, subtyping, and behavioral subtyping are different kinds of relationships. As summarized in Figure 6.5, a subclass relationship relates implementation modules, a subtype relationship relates object protocols, and a behavioral subtype relationship relates behavioral specifications. Subclasses inherit field and method declarations from superclasses, subtypes inherit interface obligations (to implement methods) from their supertypes, and behavioral subtypes inherit interface obligations and behavioral specifications from the specifications of their supertypes. Another way to look at this is in terms of the guarantees each kind

	relates	inherits	guarantees
subclass	modules	fields, methods	data format matches
subtype	object protocols	interface obligations	no type errors
beh. subtype	specifications	specifications	expected behavior

Fig. 6.5. Relationships.

of relationship makes. Roughly speaking, a subclass relationship guarantees some common data structures in objects (common field and method slots), a subtype relationship guarantees that no type errors occur when subtype objects are used in place of supertype objects, and a behavioral subtype relationship guarantees no surprising behavior occurs when subtype objects are used in place of supertype objects.

6.3 Notions of Behavioral Subtyping

Work on subtyping in type systems has important connections to behavioral subtyping. A behavioral subtype must be a subtype, since otherwise surprising behaviors (type errors) would arise. This makes sense if one thinks of structural typing as a weak behavioral specification. Two recent books describe type systems with subtyping for single-dispatch languages [AC96] and multiple-dispatch languages [Cas97].

The concept of behavioral subtyping seems to have been in the air in the late 1980s. The first edition of Meyer's book on OO software construction in Eiffel [Mey88] gives one of the first accounts of the idea. Unfortunately, Eiffel has an unsound definition of behavioral subtyping, because the language's type system has an unsound definition of subtyping [Coo89]. America gave the first sound definition of behavioral subtyping that appeared in print [Ame87] (reworked in [Ame91]); he emphasized the need for contravariance and gave a simple proof of the soundness of his rule based on Hoare's rule of consequence. The simple version of America's definition [Ame91, pp. 77–78], where the types of additional arguments and the result do not vary when a method is overridden in the subtype, uses a "transfer function" $\phi_{C\to A}$ from the abstract values of a subtype, C, to the abstract values of its supertype, A. Then it must be that:

- for each instance method g in A, g's precondition in A composed with the transfer function, $pre^g_A(\phi_{C\to A}(self\hat{}))$, must imply the corresponding precondition in C:

$$T_C \;\vdash\; \forall self\colon C\,.\,(pre^g_A(\phi_{C\to A}(self\hat{})) \Rightarrow pre^g_C(self\hat{})) \tag{6.6}$$

- for each instance method g in A, g's specification in C must be such that the following holds:

$$T_C \quad \vdash \quad \forall self\colon C \,.\, post_C^g(self\hat{\ }, self') \Rightarrow post_A^g(\phi_{C\to A}(self\hat{\ }), \phi_{C\to A}(self')) \tag{6.7}$$

The main difference between America's notion of behavioral subtyping and refinement is that it only applies to instance methods, and does not apply to class methods. America also showed how to extend the definition to deal with contravariant subtyping among the other parameters of a method and with subtyping of the result, by using the transfer functions for these arguments as well.

America's work (with van der Linden) in ECOOP/OOPSLA '90 [AvdL90] is interesting in its attempt to make behavioral subtyping statically checkable by using keywords to stand for behavioral properties.

6.3.1 Model Theory

6.3.2 For Types with Immutable Objects

Also in the late 1980s Leavens, in his Ph.D. thesis [Lea88, Lea90], showed how to use the notion of behavioral subtyping to do modular verification of OO programs [Lea91, LW90, LW95]. Leavens's definition of behavioral subtyping is model-theoretic. The basic notion is that of a coercion relation between models of abstract values [LP92], which has led to a precise model-theoretic characterization of behavioral subtyping for types with immutable objects [LP96].

Leavens's work is inspired by other model-theoretic treatments of behavioral subtyping and related ideas. A major influence is the work of Reynolds on category-sorted algebras [Rey83, Rey85]. This work forms the basis for a model-theory for multimethod languages, and a theory of subtyping based on homomorphic coercion functions (which can be generalized to homomorphic relations). The idea is that if C is a subtype of A, then there must be a coercion function from objects of type C to objects of type A, $\phi_{C\to A}$ that is preserved by the instance methods in the sense that, for example, for an instance method g that has types $A \to A$ and $C \to C$:

$$\phi_{C\to A}(g(s)) = g(\phi_{C\to A}(s)). \tag{6.8}$$

Category-sorted algebras make it possible to do modular reasoning about overloading and coercions. By generalizing static overloading to message passing (multimethod dispatch is just dynamic overloading), and coercions to subtyping, this theory also applies to OO languages with multimethods.

Another strain of model theory is based not on coercions, but on set inclusions. In such theories, the abstract values of a subtype are included in the abstract value set of each of its supertypes. Functions on such values exhibit subtype polymorphism, as a function works on any subset of its domain. Cardelli used such models in an early proof of the soundness of a type system with subtype polymorphism [Car88].

Goguen and Meseguer's Order-Sorted Algebra (OSA) [GM87], used inclusion models to help deal with subtyping in algebraic specifications. Bruce and Wegner adapted OSA to give a definition of behavioral subtyping for OO programming languages [BW90]. Such a definition says that, if one can construct a model where the set of subtype objects is a subset of the set of supertype objects, then the subtypes in question are behavioral subtypes.

The relationship between such models and models based on coercion is simple. Given a model based on coercions, one can construct an inclusion model by simply treating the set of abstract values of the supertype as the union of the sets of abstract values for their subtypes. If necessary, abstract values of each type can be "tagged" first, so that they can still be distinguished when part of a larger set. The functions that model the instance methods of the supertype can then be defined by cases, so that, for each type tag, the corresponding function from the coercion model can be run. In the other direction, one can simply take as coercion functions the inclusion function that maps each subset to its containing set.

The above construction shows that the key issue in constructing an inclusion model is not making the sets have the inclusion relationships, but in constructing the models of the instance methods.

6.3.3 Model Theory for Types with Mutable Objects

One common aspect of all the above model-theoretic approaches is that they deal with only types with immutable objects. An object is *mutable* if it has an abstract value that may vary over time.

Most OO programs contain types with mutable objects, and thus studying behavioral subtyping for such types is crucial. Though the basic idea of using a coercion relation remains valid when mutation is considered, the technical details are more complex, because mutable objects have a unique identity, and hence coercing an object from one type to another means not just creating a new value, but also associating the new value with the appropriate object identity. For this reason, Dhara's model-theoretic study [Dha97] uses coercion relations that relate not abstract values, but entire states.

Mutation also introduces the possibility of observing aliasing among objects and variables. In the presence of subtyping, one can create a state in which variables of the subtype and supertype share the same object. In such a case, if the subtype has more instance methods than the supertype, these extra methods might be able to change the shared object, when applied to the variable that has the subtype, in a way that is inconsistent with the supertype's specification [LW93b, LW94]. This problem can be dealt with in at least two ways.

Strong behavioral subtyping [LW93b, LW94] restricts the extra instance methods of a behavioral subtype to make only those state changes that are consistent with the state changes allowed by the supertype's specification. Liskov and Wing gave two different formulations of this idea. One formulation requires that each extra instance method of the subtype be supplied with a model program that shows how its effect on the abstract value can be achieved using only the instance methods of the supertype [LW93a, LW94]. Their second formulation [LW93b, LW94] requires the specification of the supertype to provide a "history constraint," which is a monotonic relation between states that says how the abstract values of that type may be changed by its instance methods; the extra instance methods of behavioral subtypes must respect the history constraint. For example, a Mutable Point type cannot be a strong behavioral subtype of an Immutable Point type, because the extra instance method that change the state of a Mutable Point would not satisfy the history constraint of Immutable Point's specification.

However, strong behavioral subtyping allows the extra instance methods of a subtype to mutate an instance's state in ways that cannot be observed through a supertype's instance methods. For example, a Triple with two immutable components and an added mutable component can be a strong behavioral subtype of an Immutable Pair type.

Strong behavioral subtyping allows all forms of aliasing, and achieves soundness of modular reasoning, because even if a supertype and subtype variable share the same subtype object, manipulations of the object through the subtype's variable cannot be surprising. Using Liskov and Wing's first formulation, this is because any mutation done by the extra instance methods can be explained by the abstract programs for the extra instance methods. Using their second formulation, this is because in reasoning about what state changes may take place one is only allowed to use the history constraint, and that must be obeyed by subtypes.

Weak behavioral subtyping [Dha97, DL95], by contrast, achieves soundness by limiting aliasing. Direct aliasing between variables of a supertype and its subtypes is prohibited. A weak behavioral subtype may have additional instance methods that change the state of a subtype object in ways that could not be explained by the supertype's instance methods, or that would violate the supertype's history constraint. Of course, the supertype's instance methods must behave similarly in the subtype.

Because of the aliasing prohibitions of weak behavioral subtyping, a type of Mutable Pairs can be a weak behavioral subtype of an Immutable Pair type. This is sound for modular reasoning because a program can only manipulate the subtype objects through variables of one of these two types, not both. Hence, if an object is being manipulated through a variable of the supertype, then it must act like an instance of the supertype, since only the common methods can be used.

Weak behavioral subtyping thus allows more subtype relationships than strong behavioral subtyping. However, the price to be paid is that the programming language used must enforce the aliasing prohibitions described above [Dha97].

6.3.4 Proof Theory

Meyer [Mey88] and America [Ame87, Ame91] both gave proof-theoretic definitions of behavioral subtyping, which were described above. Both of these definitions, however, ignored the problem of aliasing.

Cusack [Cus91] uses Z schemas in her definition of specialization, which is similar to behavioral subtyping. She does not discuss the effects of extra instance methods of the subtype on the invariants of the supertype and does not deal with aliasing.

As described above, Liskov and Wing [LW93b, LW94] were the first to offer a notion of behavioral subtyping that takes aliasing into account. Although we described it in the model theory section above, their paper actually states the definition in proof-theoretic terms.

Dhara and Leavens made only small changes to the Liskov and Wing definition in their paper that related the notions of specification inheritance and behavioral subtyping [DL96]. This idea builds on the concept of specification inheritance found in Eiffel [Mey97] and also used by Wills [Wil92] to achieve behavioral subtyping. The idea is that subtypes inherit the specifications of instance methods of their supertypes; Dhara and Leavens gave an account of specification inheritance for model-based specification languages, and proved that it ensured behavioral subtyping. They also showed how different forms of specification inheritance were needed to produce strong and weak behavioral subtypes. However, they offered no proof that the definition of behavioral subtyping used was sound with respect to some model theory.

Abadi and Leino [AL97] extend a structural type system [Car88] by behavioral specification information to the types. They present a sound axiomatic semantic semantics and provide practical guidance on reasoning about OO programs. However, their approach is not modular.

Poetzsch-Heffter and Müller give a sound Hoare-logic for a sequential subset of Java, which handles recursion, class and interface types, subtyping, inheritance, and encapsulation [PHM99]. Their work is explained further in chapter 7 of this volume.

Lewerentz and his colleagues [LLRS95] use refinement calculus for OO modeling based on observations of types. They use coercion on attributes of their language, to relate the effect constructors and methods on the states of subtype and supertype objects. They do not consider aliasing or interference.

Utting [UR92, Utt92] defined behavioral subtyping using the refinement calculus. The refinement calculus offers a way to prove behavioral subtyping in this setting. His definition does not, however, allow for change of data representations.

The work of Mikhajlova and her coauthors [BMvW97, MS97, Mik98b] allows the sound verification of OO programs in a refinement calculus framework. The key concept is that of class refinement (called *correct subclassing*) which (as described above) is stronger than behavioral subtyping, since it involves class methods. Class refinement, in addition to providing the same guarantee against surprising behavior when objects of subclasses are manipulated, also allows one to verify programs that use class methods (and even expressions denoting class objects) to create new instances. However, treating subclasses as subtypes and behavioral subtypes collapses the distinctions shown in Figure 6.5. This restricts both subclass and subtype relationships. For example, treating subclasses as subtypes restricts the use of binary methods [BCC+95]. Conversely, treating subclasses as behavioral subtypes limits certain uses of inheritance; for example, Doubly-Ended Queue could otherwise be a subtype of Stack, even though it is more convenient for Stack to inherit from Doubly-Ended Queue [Sny86].

6.4 Behavioral Subtyping for Components

A common theme in work on behavioral subtyping is that objects of behavioral subtypes should be able to be manipulated without surprises, where surprises are defined relative to the specification of the supertype. Hence this method of modular reasoning method is called *supertype abstraction* [LW95]. Therefore, if we wish to reason about the correctness of component-based systems using supertype abstraction, the key issue is the notion of behavioral subtyping for such systems. To sketch this, we propose looking at their specifications and making an analogy to OO programs.

In Section 6.1.2, we saw that, in general, model programs were needed to fully specify components. We can also consider the usual pre- and post-condition style specifications to be a special case of model programs, since such a specification can be considered to be a model program with a single specification statement.

Our approach for defining behavioral subtyping is based on refinement of these model programs. For both strong and weak behavioral subtyping the key idea is that the common instance methods of the subtype and each supertype must be such that each such method's model program in the specification of the subtype refines its specification in that supertype. This, and requirements that the invariant and history constraint (for the common methods) of the subtype imply those in each supertype, is enough for weak behavioral subtyping. For example, the type `Directory` is a weak behavioral subtype of `RODirectory`. (We hope to formally relate this to an extension of the refinement calculus [Mor94, MV94, Woo91] as future work.)

For strong behavioral subtyping, one approach is to require that the history constraint (for all the methods) of the subtype must imply that of each supertype [DL96, LW93b, LW94]. This limits what the extra methods can do in terms of mutation of the objects of the subtype, but does not place any limits on what events they may signal. Thus Liskov and Wing's other approach, of requiring a program that explains the effect of each additional instance method of the subtype in terms of the supertype's methods [LW93a, LW94], has more promise. Indeed, the refinement calculus paradigm makes it clear what must be verified to prove strong behavioral subtyping; that is, that for each additional instance method, m, of the subtype, there must be some model program, p_m, such that p_m is expressed only using the methods of the supertype, and the specification of m refines p_m.

Using the second form of strong behavioral subtyping, the type `Directory` is not a strong behavioral subtype of the type `RODirectory`, because the type `RODirectory` has a mapping, from names to files that is visible to clients, but this mapping is modified by the instance methods of `Directory`. However, `RODirectory` is a strong behavioral subtype of `DirObserverKeeper`, as `RODirectory` just adds to the model fields of `DirObserverKeeper`.

6.5 Conclusions

In this chapter we have discussed the specification of component-based systems. We noted that a combination of model variables [Lei95] and model programs (as in the refinement calculus) seem adequate for specification of callbacks that occur in such systems [BS97]. Our notion of behavioral subtyping for components is based on these specifications, in that we require that behavioral subtypes obey the specifications of the instance methods of their supertypes. We sketched both weak and strong behavioral subtyping.

Clearly we have only given a sketch of what behavioral subtyping should be for component-based systems. Much work remains in fleshing out the details and proving that such notions permit sound modular reasoning.

Acknowledgements

Thanks to the anonymous referees for comments on an earlier draft of this paper. Thanks to Al Baker for discussions about the JML examples used in this paper. Leavens's work was supported in part by the United States of America's National Science Foundation under Grant CCR-9803843.

Bibliography

[AC96] Abadi, M. and Cardelli, L. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, N.Y., 1996.

[AL97] Abadi, M. and Leino, R. A logic of object-oriented programs. In Bidoit, M. and Dauchet, M., editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, New York, N.Y., 1997.

[Ame87] America, P. Inheritance and subtyping in a parallel object-oriented language. In Bezivin, J. et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, New York, N.Y., June 1987. Springer-Verlag. Lecture Notes in Computer Science, Volume 276.

[Ame91] America, P. Designing an object-oriented programming language with behavioural subtyping. In de Bakker, J. W., de Roever, W. P., and Rozenberg, G., editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, N.Y., 1991.

[AvdL90] America, P. and van der Linden, F. A parallel object-oriented language with inheritance and subtyping. *ACM SIGPLAN Notices*, 25(10):161–168, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[Bac88] Back, R. J. R. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, August 1988.

[BCC+95] Bruce, K., Cardelli, L., Castagna, G., Group, T. H. O., Leavens, G. T., and Pierce, B. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[BJ95] Broy, M. and Jähnichen, S., editors. *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Computer Science*, New York, N.Y., 1995. Springer-Verlag.

[BMvW97] Back, R., Mikhajlova, A., and von Wright, J. Class refinement as semantics of correct subclassing. Technical Report 147, Turku Centre for Computer Science, December 1997. http://www.tucs.abo.fi/publications/techreports/TR147.html.

[BS97] Büchi, M. and Sekerinski, E. Formal methods for component software: The refinement calculus perspective. In *Proceedings of the Second Workshop on Component-Oriented Programming (WCOP)*, June 1997. ftp://ftp.abo.fi/pub/cs/papers/mbuechi/FMforCS.ps.gz.

[Bv92] Back, R. J. R. and von Wright, J. Combining angels, deamons and miracles in program specifications. *Theoretical Computer Science*, 100(2):365–383, June 1992.

[BvW98] Back, R.-J. and von Wright, J. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.

[BW90] Bruce, K. B. and Wegner, P. An algebraic model of subtype and inheritance. In Bançilhon, F. and Buneman, P., editors, *Advances in Database Programming Languages*, pages 75–96. Addison-Wesley, Reading, Mass., August 1990.

[Car88] Cardelli, L. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 51–66.

[Cas97] Castagna, G. *Object-Oriented Programming: A Unified Foundation.* Progress in Theoretical Computer Science. Birkhauser, Boston, 1997.

[Coo89] Cook, W. R. A proposal for making eiffel type-safe. *The Computer Journal*, 32(4):305–311, August 1989.

[Cus91] Cusack, E. Refinement, conformance, and inheritance. *Formal Aspects of Computing*, 3:129–141, January 1991.

[Dha97] Dhara, K. K. Behavioral subtyping in object-oriented languages. Technical Report TR97-09, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames IA 50011-1040, May 1997. The author's Ph.D. dissertation.

[Dij76] Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976.

[DL95] Dhara, K. K. and Leavens, G. T. Weak behavioral subtyping for types with mutable objects. In Brookes, S., Main, M., Melton, A., and Mislove, M., editors, *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. http://www.elsevier.nl/locate/entcs/volume1.html.

[DL96] Dhara, K. K. and Leavens, G. T. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

[DW99] D'Souza, D. F. and Wills, A. C. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Object Technology Series. Addison Wesley, Reading Mass., 1999.

[EHL+94] Edwards, S. H., Heym, W. D., Long, T. J., Sitaraman, M., and Weide, B. W. Part ii: Specifying components in RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):29–39, Oct 1994.

[EHMO91] Ernst, G. W., Hookway, R. J., Menegay, J. A., and Ofgen, W. F. Modular verification of Ada generics. *Computer Languages*, 16(3/4):259–280, 1991.

[FGJM85] Futatsugi, K., Goguen, J. A., Jouannaud, J.-P., and Meseguer, J. Principles of OBJ2. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, January 1985.

[FJ92] Feijs, L. M. G. and Jonkers, H. B. M. *Formal Specification and Design*, volume 35 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, UK, 1992.

[GHG+93] Guttag, J. V., Horning, J. J., Garland, S., Jones, K., Modet, A., and Wing, J. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, N.Y., 1993.

[GHJV95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Mass., 1995.

[GM87] Goguen, J. A. and Meseguer, J. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. In *Symposium on Logic in Computer Science, Ithaca, NY*, pages 18–29. IEEE, June 1987.

[GM94] Gardier, P. H. B. and Morgan, C. A single complete rule for data refinement. In Morgan and Vickers [MV94], pages 111–126.

[Gog84] Goguen, J. A. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[Hes92] Hesselink, W. H. *Programs, Recursion, and Unbounded Choice*, volume 27 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, New York, N.Y., 1992.

[Hoa69] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Hoa72] Hoare, C. A. R. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[Jon90] Jones, C. B. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[LBR99] Leavens, G. T., Baker, A. L., and Ruby, C. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06f, Iowa State University, Department of Computer Science, July 1999.

[Lea88] Leavens, G. T. *Verifying Object-Oriented Programs that use Subtypes.* PhD thesis, Massachusetts Institute of Technology, December 1988. Published as MIT/LCS/TR-439 in February 1989.

[Lea90] Leavens, G. T. Modular verification of object-oriented programs with subtypes. Technical Report 90-09, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, July 1990. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[Lea91] Leavens, G. T. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[Lei95] Leino, K. R. M. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[LG86] Liskov, B. and Guttag, J. *Abstraction and Specification in Program Development.* The MIT Press, Cambridge, Mass., 1986.

[LLRS95] Lewerentz, C., Lindner, T., Rüping, A., and Sekerinski, E. On object-oriented design and verification. In Broy and Jähnichen [BJ95], pages 92–111.

[LP92] Leavens, G. T. and Pigozzi, D. Typed homomorphic relations extended with subtypes. In Brookes, S., editor, *Mathematical Foundations of Programming Semantics '91*, volume 598 of *Lecture Notes in Computer Science*, pages 144–167. Springer-Verlag, New York, N.Y., 1992.

[LP96] Leavens, G. T. and Pigozzi, D. An exact algebraic characterization of behavioral subtyping. Technical Report 96-15, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, November 1996. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[LP97] Leavens, G. T. and Pigozzi, D. The behavior-realization adjunction and generalized homomorphic relations. *Theoretical Computer Science*, 177:183–216, 1997.

[LW90] Leavens, G. T. and Weihl, W. E. Reasoning about object-oriented programs that use subtypes (extended abstract). In Meyrowitz, N., editor, *OOPSLA ECOOP '90 Proceedings*, volume 25(10) of *ACM SIGPLAN Notices*, pages 212–223. ACM, October 1990.

[LW93a] Liskov, B. and Wing, J. M. A new definition of the subtype relation. In Nierstrasz, O. M., editor, *ECOOP '93 — Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany*, volume 707 of *Lecture Notes in Computer Science*, pages 118–141. Springer-Verlag, New York, N.Y., July 1993.

[LW93b] Liskov, B. and Wing, J. M. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, October 1993. *OOPSLA '93 Proceedings*, Andreas Paepcke (editor).

[LW94] Liskov, B. and Wing, J. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[LW95] Leavens, G. T. and Weihl, W. E. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.

[Mey88] Meyer, B. *Object-oriented Software Construction.* Prentice Hall, New York, N.Y., 1988.

[Mey92] Meyer, B. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[Mey97] Meyer, B. *Object-oriented Software Construction.* Prentice Hall, New York, N.Y., second edition, 1997.

[MG90] Morgan, C. and Gardiner, P. H. . B. Data refinement by calculation. *Acta Informatica*, 27(6):481–503, May 1990.

[Mik98a] Mikhajlova, A. Consistent extension of components in presence of explicit invariants. In Weck, W., Bosch, J., and Szyperski, C., editors, *Third International Workshop on Component-Oriented Programming (WCOP'98) held in conjunction with ECOOP'98.* TUCS General Publication Series, No. 10, July 1998.

[Mik98b] Mikhajlova, A. Refinement of generic classes as semantics of correct polymorphic reuse. In *Proceedings of the International Refinement Workshop, and Formal Methods Pacific (IRW/FMP'98)*, Springer Series in Discrete Mathematics and Theoretical Computer Science, pages 266–285, New York, N.Y., Jul 1998. Springer-Verlag.

[Mor87] Morris, J. M. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[Mor89] Morris, J. M. Laws of data refinement. *Acta Informatica*, 26(4):287–308, February 1989.

[Mor90] Morgan, C. *Programming from Specifications.* Prentice Hall International, Hempstead, UK, 1990.

[Mor94] Morgan, C. *Programming from Specifications: Second Edition.* Prentice Hall International, Hempstead, UK, 1994.

[MS97] Mikhajlova, A. and Sekerinski, E. Class refinement and interface refinement in object-oriented programs. In Fitzgerald, J., Jones, C. B., and Lucas, P., editors, *FME '97: Industrial Applications and Stengthened Foundations of Formal Metohds*, volume 1313 of *Lecture Notes in Computer Science*, pages 82–101, NY, 1997. Springer-Verlag.

[MSL99] Mikhajlov, L., Sekerinski, E., and Laibinis, L. Developing components in the presence of re-entrance. Technical Report TUCS-TR-239, TUCS - Turku Centre for Computer Science, February 1999.

[MV94] Morgan, C. and Vickers, T., editors. *On the refinement calculus.* Formal approaches of computing and information technology series. Springer-Verlag, New York, N.Y., 1994.

[Nip86] Nipkow, T. Non-deterministic data types: Models and implementations. *Acta Informatica*, 22(16):629–661, March 1986.

[OSWZ94] Ogden, W. F., Sitaraman, M., Weide, B. W., and Zweben, S. H. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, Oct 1994.

[PHM99] Poetzsch-Heffter, A. and Müller, P. A programming logic for sequential Java. In Swierstra, S. D., editor, *European Symosium un Programming (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

[Rey83] Reynolds, J. C. Types, abstraction and parametric polymorphism. In *Proc. IFIP Congress '83, Paris*, September 1983.

[Rey85] Reynolds, J. C. Three approaches to type structure. In Ehrig, H., Floyd, C., Nivat, M., and Thatcher, J., editors, *Mathematical Foundations of Software Development, Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin. Volume 1: Colloquium on Trees in Algebra and Programming (CAAP '85)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer-Verlag, New York, N.Y., March 1985.

[SBC92] Stepney, S., Barden, R., and Cooper, D., editors. *Object Orientation in Z.* Workshops in Computing. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

[Sch90] Schoett, O. Behavioural correctness of data representations. *Science of Computer Programming*, 14(1):43–57, June 1990.

[Sny86] Snyder, A. Encapsulation and inheritance in object-oriented programming languages. *ACM SIGPLAN Notices*, 21(11):38–45, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[SWO97] Sitaraman, M., Weide, B. W., and Ogden, W. F. On the practical need for abstraction relations to verify abstract data type representations. *IEEE Transactions on Software Engineering*, 23(3):157–170, March 1997.

[Szy98] Szyperski, C. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, N.Y., 1998.

[UR92] Utting, M. and Robinson, K. Modular reasoning in an object-oriented refinement calculus. In Bird, R. S., Morgan, C. C., and Woodcock, J. C. P., editors, *Mathematics of Program Construction, Second International Conference, Oxford, U.K., June/July*, volume 669 of *Lecture Notes in Computer Science*, pages 344–367. Springer-Verlag, New York, N.Y., 1992.

[Utt92] Utting, M. *An Object-Oriented Refinement Calculus with Modular Reasoning.* PhD thesis, University of New South Wales, Kensington, Australia, 1992. Draft of February 1992 obtained from the Author.

[Wil92] Wills, A. Specification in Fresco. In Stepney et al. [SBC92], chapter 11, pages 127–135.

[Win83] Wing, J. M. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[Woo91] Woodcock, J. C. P. A tutorial on the refinement calculus. In Prehn, S. and Toetenel, W. J., editors, *VDM '91 Formal Software Development Methods 4th International Symposium of VDM Europe Noordwijkerhout, The Netherlands, Volume 2: Tutorials*, volume 552 of *Lecture Notes in Computer Science*, pages 79–140. Springer-Verlag, New York, N.Y., October 1991.