# STRENGTH AND ESSENTIALITY OF SPECIFICATIONS

Jeannette M. Wing

Computer Science Department
Henry Salvatori Computer Science Center
University of Southern California
Los Angeles, CA 90089

7 October 1983

Formal specifications can and should play an important role in the design phase of the programming process. Uncovering design flaws early can save some of the time, effort, and resources spent in uncovering them later in testing and debugging phases. Forcing oneself to be precise helps clarify and disambiguate a client's problem statement. Often it is the act of specifying, and not the final product that is most useful in program design. A specification also serves as a valuable piece of documentation.

One of the problems specifiers face when writing a formal specification for a large piece of software is that no feedback is provided to indicate whether the specification is on the "right track." One method of attacking this problem is to provide ways to evaluate a specification as it incrementally develops.

We discuss two ways that a specifier can evaluate a specification. One is to compare two specifications with respect to their relative strength (Section 2). The second is to determine parts of a specification that are inessential to show that some desired property holds (Section 3). Another way, which we do not discuss in this paper but which is addressed in detail in [Wing],[1] is to check a specification for certain properties, e.g., consistency and completeness. We give some background information in Section 1 to establish our context. [Wing] contains further details and precise definitions.

## 1. Background

We treat a specification as a formal system that specifies a theory. A *formal system*, Spec = <L, A, R>, consists of a language L (a set of symbols and a set of well-formed formulae), a set of axioms A, and a set of rules of inference R. A *theory* specified by a formal system, Spec, is the smallest set of formulae reflexively and transitively closed over the set of axioms under the rules of Spec. The theory we associate with a specification is a first-order theory with equality.

The specification language we use is targeted for the CLU programming language. We are concerned with specifying procedures and clusters, and thus, with defining the theories of specifications of procedures and clusters. A cluster names a type and defines a set of procedures that create or manipulate objects of that type. An object whose value can change is said to be *mutable*.

The formulae in the theory of a procedure specification, Pr, are triples, P{Pr}Q, where P is a first-order assertion on the initial values of objects, and Q is a first-order assertion on the initial and final values of objects. The theory of a cluster specification is the union of the theories of its procedure specifications plus the set of triples derivable from a type induction rule associated with a cluster specification. There is a hypothesis in the rule for each procedure specified to create or mutate an object of that type. The type induction rule lets us associate a type invariant with a cluster specification.

We focus on expressing our definitions in terms of theories, and not of models, because a theory can be viewed as a set of formulae, i.e., strings of symbols, subject to syntactic manipulation. Given sufficient mechanized theorem-proving power, we can reason about specifications independently of their models by just manipulating their theories as we do their text.

## 2. Comparing Strength

Intuitively, the stronger a specification, the fewer the number of implementations that satisfy it. One situation in which it is useful to know when a specification is *as strong as* another is in that ensuring the strength of a specification is unchanged after a modification to it is made. For example, if we rename identifiers of a specification in order to have mnemonic names, we would want to make sure we have made only a syntactic and not a semantic change. A second situation is in determining if it is permissible to replace a specification with another without affecting any of its users. If one specification is as strong as another, then under certain circumstances we should be able to substitute one for other.

Sometimes, we may want a *stronger* specification. We might realize the specification is not strong enough in trying to prove a property of the specification or of what we intend it to specify. We revisit

---

[1] Wing, J.M., "A Two-Tiered Approach to Specifying Programs," Laboratory for Computer Science TR-299, MIT Cambridge, MA, June 1983. Also Ph.D. thesis, Dept. of EE and CS, MIT, May 1983.

this situation when we discuss essentiality of a specification in Section 3. Finally, if we were to decide to strengthen a specification, we might want to compare the new and original specifications to make sure we did not make them *incomparable.*

### 2.1. Definition of Strength

To define strength formally, we borrow the analogous concept from logic that the stronger a theory, the fewer the number of models that satisfy it, and define a *strength* relation between specifications in terms of strength between their theories. For example, the theory of <Z, +, -> is as strong as <N, 0, succ>, but not vice versa, where Z is the set of all integers, and N is the set of all natural numbers. Determining when one theory is as strong as another depends on finding an *interpretation* that translates formulae (not just symbols) of one theory into those of another. Most of the following definitions are adapted from [Enderton].[2] Let Th1 be a theory in a language L1 and Th2 be a theory in a possibly different language L2.[3] Let $\pi$ be a mapping from L1 into L2.

Def: If $\forall\sigma \in$ L1 $[\sigma \in$ Th1 $\Rightarrow \pi(\sigma) \in$ Th2], then $\pi$ is an *interpretation of Th1 into Th2.*

Def: Th1 is *as strong as* Th2 if there exists an interpretation of Th2 into Th1.

Def: Th1 is *stronger* than Th2 if Th1 is as strong as Th2 and Th2 is not as strong as Th1.

Def: Th1 and Th2 are *incomparable* if Th1 is not as strong as Th2 and Th2 is not as strong as Th1.

We extend the last three definitions to two specifications in the obvious way. For example, given two specifications, Spec1 and Spec2, Spec1 is as strong as Spec2 if Th(Spec1) is as strong as Th(Spec2).

### 2.2. Applying the Definition to Cluster Specifications

It would be useful to characterize changes we can make to a specification by their effect on the strength of the original specification. For example, adding a procedure specification to a cluster specification might strengthen the cluster specification, or it might not. Depending on what kind of procedure specification is added can restrict the possible effects on the strength of the cluster specification. Let us consider two kinds of procedure specifications: *constructors* and *observers.* A constructor of type T specifies that a procedure return or mutate objects of type T; an observer of type T specifies that a procedure return or mutate objects of type other than T.

Adding a constructor has the possible effect of leaving the original specification unchanged, making it incomparable to the new, or weakening it. We conjecture that adding a constructor cannot strengthen a cluster specification because adding a constructor adds a hypothesis to the type induction rule. Adding a hypothesis to the rule might leave it unchanged, weaken it, or invalidate it; only by deleting a hypothesis can we possibly conclude a stronger invariant.

Here is an example. Let Spec1 be a *set* cluster specification with the sole constructors *singleton* and *union* and the type invariant that all sets are of size strictly greater than zero. Suppose we add a *pair* constructor. Since formulae involving *pair* can be expressed in terms of *singleton* and *union,* no theorems of Th(Spec1) are invalidated and no new theorems are added. This supports our intuition that adding a constructor that does not change the type invariant should not strengthen a cluster specification. Suppose, instead that we add to Spec1 a *create* constructor to make Spec2. One might think that by the addition of *create,* Th(Spec2) would be strictly larger than Th(Spec1) and so Th(Spec2) would be stronger than Th(Spec1). This is not true, however, since the property that all sets are of size greater than zero is true of Spec1, but not of Spec2, and the property that there exists a set of size equal to zero is true of Spec2, but not of Spec1. This example illustrates a perhaps surprising consequence of our definition since, intuitively, we would think that adding a constructor that increases the value set of a type should strictly strengthen a cluster specification.

---

[2] Enderton, H. B., *A Mathematical Introduction to Logic,* Academic Press, New York, 1972.

[3] L2 must include equality for technical reasons.

Adding an observer can strengthen a cluster specification or leave it unchanged. Since hypotheses of the type induction rule deal with only constructors, adding an observer has no effect on the type induction rule of the cluster specification. Hence, the addition of an observer cannot weaken or invalidate the type induction rule. As an example of strengthening with an observer, consider adding a *size* observer to a *stack[elem]* cluster specification that has only constructors. Doing so adds theorems about integers to the Th(*stack[elem]*). As an example of leaving the strength unchanged, suppose *stack[elem]* has *null*, *push*, and *top*, where *top* mutates its stack argument. Adding a *read* observer that is like *top* except that it does not mutate its stack argument, does not change the strength of the original specification.

## 3. Essentiality

In the construction of a specification, we often want it to be "minimal" in a given context. That is, we would like to able to pare down a specification to just the "essential part" necessary for a desired set of properties to hold. Removing parts that have been shown to be inessential gives us a way of paring down a specification. Henceforth, we limit our discussion on essentiality to the converse notion of "inessentiality."

A part, P, of a specification, Spec, is *inessential* for a theory, T, if Spec with P removed can still be used to deduce the theorems in T. We say "P is an inessential part of Spec for T." Identifying a part of a specification that is inessential to prove a property means that we can freely remove or alter that part of the specification and still be ensured that the desired property holds. On the other hand, if we were to change some part that is essential then we might have to reverify that the property holds.

### 3.1. Definition

For a specification, Spec = <L, A, R>, L is its language, A is its set of axioms, and R is its set of rules. Let T be a theory such that each formula in T is deducible from Spec. We write this "Spec $\vdash$ T."

Def: A *part* of Spec is a specification with a language, L' $\subseteq$ L, a set of axioms, A' $\subseteq$ A, and a set of rules, R' $\subseteq$ R.

Def: Let P = <L', A', R'> be a part of Spec. (Spec - P) is the specification whose language is (L - L'), whose set of axioms is (A - A'), and whose set of rules is (R - R').

Def: *P is an inessential part of Spec for T* if and only if (Spec - P) $\vdash$ T.

Checking for inessentiality must be done with respect to a theory since a part of a specification that is inessential for one theory might be essential for a different theory.

### 3.2. Applying the Definition

Here are two situations in which it would be useful to determine whether a part of a specification is inessential. One situation is to determine what part of a cluster specification is inessential to prove the type invariant. For example, theorems dealing only with observers are inessential. Theorems dealing with some of the constructors may be inessential as well, if removing them does not change the strength of the cluster specification. A second situation is to determine what part of a specification is inessential in the proof of satisfaction between an implementation, Imp, and a specification, Spec. Suppose in the proof we use a specification S, whose theory is a subset of Th(Imp). In knowing what part of S is inessential to the proof of satisfaction, we can change that part of S and be guaranteed that Imp still satisfies Spec.

## 4. Summary

We presented two ways to evaluate specifications by defining what we mean by the *strength* of a specification and an *inessential* part of a specification. What remains is to develop algorithms and build software tools based on these algorithms that enable us to compare two specifications with respect to their strength and to test whether parts of a specification are inessential or not. A specifier can then use these software tools to help evaluate a specification as it develops.

WORKSHOP NOTES

# INTERNATIONAL WORKSHOP ON MODELS AND LANGUAGES FOR SOFTWARE SPECIFICATION AND DESIGN

## March 30 1984

**Orlando, Florida**

EDITED BY

Robert G. Babb II
Oregon Graduate Center

Ali Mili
Université Laval

Sponsored by: IEEE COMPUTER SOCIETY

UNIVERSITÉ LAVAL

LCRST OF JAPAN

ogc
OREGON GRADUATE CENTER

In cooperation with: acm ACM SIGSOFT