Critique of the "Lake Arrowhead Three"

Leslie Lamport
Digital Equipment Corporation

20 March 1992

1 The Rules of the Game

This critique compares the three specification methods presented in this issue, based on their solutions to the database serializability problem. I have tried to judge the methods rather than the papers, though it is often hard to separate pedagogical problems from methodological ones. To ensure that my comments were based on a proper understanding of their work, I corresponded with the authors before writing this. I have also, on previous occasions, met with Kurki-Suonio and Broy and discussed their work. For compactness, the three papers are referred to as *Br* (for Broy), *L&S* (Lam and Shankar), and *K-S* (Kurki-Suonio).

The methods are compared on seven criteria. While neither complete nor disjoint, these criteria provide a convenient framework for the comparison. A single article cannot discuss all aspects of specification, and none of the papers provide a user's manual for a complete specification method. In pointing out omissions, my purpose is to elucidate what is and is not discussed, not to criticize the authors.

2 Fidelity to the Problem

To compare different formal specifications, we must first examine what they are specifying. All three papers were supposed to specify the same thing; so, we should see if they did. But first, a discussion of what they were supposed to specify is in order.

I chose the database serializability problem partly because it is well known and I thought it had the right degree of complexity. But what really attracted to me to it was that other specification methods handled it better than the method I was then advocating. (Knowledge comes from examining one's failures, not from displaying one's successes.)

The traditional definition of serializability is most easily expressed in terms of execution histories. I use an *assertional* approach to verification, where one reasons about the current state. Assertional reasoning about histories requires adding a dummy variable to record the execution history. I found such history variables philosophically unappealing, because I believed the state should include only information that must be there in a real implementation. When I tried to specify serializability in terms of the "real" current state, the specification became extremely complicated.

I am no longer bothered by this problem. After posing the problem, I discovered how to write a specification without dummy variables that I found satisfactory. A specification for a system in which transactions consist of only a single read-and-modify command appears in [7], and it can be extended to the Lake Arrowhead problem. Moreover, I am now reconciled to the need for dummy variables. We have learned that to be complete, assertional methods for reasoning about specifications must allow dummy variables (or their equivalent)—not only history variables for recording past behavior, but *prophecy variables* for predicting future behavior [1]. If a method lacks either class of dummy variables, there will be correct implementations of a specification that the method cannot verify. Logical necessity is a good antidote to philosophy.

But the main reason I am no longer bothered by the problem is that I now regard it as a bad problem. The usual notion of serializability, which requires only that the values returned by a transaction be consistent with some serial execution of the transactions, is too weak. For example, it permits an implementation to throw away all values written by any completed transaction that only writes and does not read. The resulting execution is serializable, producing the same results as one in which all such write-only transactions are done last. A more realistic notion of serializability requires that if one transaction commits before another begins, then the first transaction appears before the second in the serialization order. The difficulty I encountered specifying the naive definition of serializability stemmed from the problems with that definition. The more realistic form of serializability can be specified assertionally without using a variable to record the entire execution history.

So, in retrospect, the problem is an unfortunate one. A formal specification should not be judged harshly because it fails faithfully to formalize an unreasonable informal specification.[1] In any case, the two implementations in the problem statement are quite reasonable, and the inability to accurately specify either one of them would indicate an inadequacy of the method.

2.1 *K-S*

Serializability is achieved by having the system magically predict, at the beginning of a transaction, the transaction's position in the final serialization order. Since the predicted value is an internal parameter that is not revealed until much later, no real ability to predict the future is required. This approach is similar to the one I used in [7]. However, while it provides a reasonable specification, it is not as general as the informal notion of serializability. Because the transaction's position is chosen once and for all, the specification does not allow an implementation in which write-only transactions are deferred forever.[2]

The specifications of the implementations are not carried out to the level stated by the problem. The implementations should describe actual programs that call the lower-level database, together with a specification of the lower-level database. Instead, *K-S* writes intermediate-level specifications that require an additional refinement to insert the lower-level procedure calls. This refinement is fairly simple, but it must be taken into account when comparing the three papers.

2.2 *L&S*

The high-level specification is simple and appealing. A history variable records the entire sequence of operations, and the informal concept of serializability is defined in a straightforward fashion as a condition on the value of that variable. The specification does introduce extraneous transaction

[1] However, I disagree with the criticism of the problem in Section 2 of *Br*. In the informal definition of the *read* operation, the value must be "current" with respect to the serialization order. Although Broy's incorrect interpretation is completely understandable under the circumstances, I believe the intended meaning would have been clear to him had he been more familiar with the concept of serializability.

[2] Kurki-Suonio claims that such an implementation is not correct, because it permits no serialization of the complete set of transactions resulting from an infinite execution. However, I believe that most people would regard serializability, as specified informally in the problem statement, to be a safety property—one that constrains only finite prefixes of an infinite behavior—and would therefore disagree with him.

identifiers, which were not part of the informal specification. However, this appears to be a mistake on the part of the specifiers, not indicative of any shortcoming of their method. Lam and Shankar could have written a simpler specification in their method without transaction identifiers.

The implementations are specified essentially as stated in the problem. I found only one minor discrepancy between the informal and formal specifications, in the two-phase commit protocol.

2.3 *Br*

The high-level specification apparently does capture the intuitive notion of serializability in full generality. However, as discussed below, I found the specification too hard to understand for me to be sure exactly what it specifies. There is also at least one minor difference between the formal specification of the interface and the informal description.

The implementations are not specified as required. Instead of specifying an implementation of the two-phase commit protocol, *Br* strengthens the high-level specification to one that would be satisfied by such an implementation. The timestamp implementation is not specified at all. While the implementations can undoubtedly be specified with the paper's method, *Br* provides no indication of how formidable a task that would be.

3 Simplicity of the Specification

A formal specification is meant to be read by human beings, so it is natural to ask how easy it is for them to read and understand the specifications.

3.1 *K-S*

Today, everyone is first introduced to computers through a programming language. Because *K-S*'s specifications are written in a programming language based style, most computerists will find them easy to read. Moreover, the state-chart descriptions are quite appealing. Similar pictures can be drawn for most methods. However, if they are drawn by hand, there is no way to ensure that the pictures correspond to the formal text of the specification. Kurki-Suonio informs me that, although the statecharts in the paper were drawn by hand, tools now exist for deriving them automatically from the specification.

It is not hard to see that the specification implies serializability. However, it is not obvious that it is equivalent to serializability, which is not surprising, since we have seen that it isn't. (It rules out implementations that throw away write-only transactions.) However, Kurki-Suonio could have written a high-level specification similar to that of *L&S*, using a history variable.

3.2 *L&S*

Although Lam and Shankar describe a method for writing formal specifications, the specifications they write are not really formal. The problem is largely syntactic. Explanation and formalism are not separated. It is impossible to take a pair of scissors to their paper, cut away the informal prose, and be left with a complete, formal specification.

It should not be hard to write a precise syntax for specifications in this method, at the level of detail in which the DisCo language is defined in *K-S*. Separating formalism from comments would tend to make the specifications easier to read. However, a formal language would undoubtedly rule out some informal notation that was used—for example, the convention of dropping components from a tuple and inferring the complete tuple from the names would not be allowed in any precisely defined specification language. Without seeing a more formal version of the specification, it is hard to judge the readability of formal specifications written with the method.

3.3 *Br*

I found this specification the hardest to understand, largely for a very surprising reason. In the method of *Br*, a system is specified by defining the sequence of actions that can be produced when the system is executed. The method would seem naturally to lead to a specification of serializability in terms of this sequence. Instead, *Br* specifies serializability in terms of states, confirming what I had long known—a specification of serializability in terms of the database state is very difficult.

Considering what Broy was trying to do, it is not surprising that his specification is so hard to read. The bizarre method of allowing the reuse of keys does not help matters. Any comparison of this specification with the ones in the other papers is bound to be misleading, since the other authors chose an easier route.

4 Simplicity of the Semantics

Simplicity of a specification can be deceiving. One can write simple-looking specifications consisting of pretty pictures, full of friendly bubbles and arrows, that are simple and easy to read, but are devoid of formal meaning. True simplicity of a formal specification requires that the formal semantics of the specification language be simple. The real complexity of a specification must take into account the difficulty in understanding its formal meaning.

4.1 *K-S*

A possible execution of the system is described as a sequence of externally-observable state changes, specified by a state/transition system with fairness conditions on the transitions. The correspondence between the language's action definitions and the semantic state transitions is straightforward, making it fairly easy to understand how the specification describes a sequence of state changes.

One hidden complexity in the translation from language to semantics is caused by the use of objects. The use of objects essentially hides an extra parameter—the object identity—from most formulas. To understand the meaning of the formula, one must include that parameter. This is a simple translation (though not clearly explained in *K-S*), and poses no real problem in understanding a specification. On the other hand, making the parameter explicit would not have made the specification significantly harder to read in the first place. Moreover, the inability to refer to the implicit parameter can sometimes complicate a specification. (In one small example used by Kurki-Suonio to illustrate his method, the specification can be simplified by dispensing with objects and working directly with arrays.) I expect that the use of objects, while appealing to the current fad for object-orientation, makes little practical difference.

4.2 *L&S*

A possible execution of the system is described as a sequence of actions, specified by a state/transition system, either augmented with fairness conditions on the transitions, or with additional invariance and liveness properties. While each of the concepts is simple, understanding is made more difficult by the use of different ways of specifying the same thing.

There are two specific instances of redundancy in the method. The first is the distinction between a module and an interface. There is no fundamental reason for having both modules and interfaces; semantically, they are both just sets of sequences of actions. This distinction apparently comes from the area of communication protocols, interfaces corresponding to "service" specifications and modules corresponding to "protocol" specifications. The second redundant concept is the invariance assumption/guarantee in an interface specification. Such an invariance property can be expressed by the state/transition system. Still, the complexity introduced by these redundant concepts is not great.

4.3 *Br*

A possible system execution is described as a sequence of actions, specified by a stream function. The meaning of the stream functions is defined by the simple, elegant semantics of algebraic specification. The translation from precise specification language to semantics is as simple as any formal method is likely to achieve.

Readers who have been intimidated by the formidable mathematics employed in the literature on algrebraic specification might find my belief in its simplicity surprising. However, the difficulty of that literature arises because it addresses fundamental problems that are ignored by much work on specification, including *K-S* and *L&S*. Despite their formal framework, neither *K-S* nor *L&S* considers how all the functions introduced in their specifications are formally defined. They assume definitions of such things as set operations, and they do not describe precisely how one defines new operations. It is quite proper for them to ignore these problems in order to concentrate on issues particular to concurrent systems. However, the problems must be solved to provide a specification method with a rigorous formal semantics. *Br*, unlike *K-S* and *L&S*, contains specifications with completely defined formal semantics.

5 Help in Structuring the Proof

The problem statement required proofs that two given implementations meet the specification. "Proof" is a word that admits of many interpretations—from the informal sketches that commonly appear in mathematical journals, to complete mechanical verification. Informal proofs are notoriously unreliable. The "social process" of mathematics does not operate in the world of system design [5], and viable formal methods must provide their users with some way of systematizing proofs, allowing them to break large proofs into small, easily checked pieces.

5.1 *K-S*

No formal proof method is presented. Instead, the proof is carried out at the semantic level of sequences of states. Such proofs are at best of limited value, since a possibility that is overlooked in writing the specification is likely to be overlooked when writing the proof.

At the end of Section 2.1, it is hinted that DisCo can be formalized in terms of Back's refinement calculus [2], which does provide a formal proof method. However, as explained below, the incompleteness of Back's proof system makes it incapable of verifying the implementations in this example. Moreover, Back's method requires a translation of liveness properties, involving the introduction of counter variables, and it is not clear if it will be practical for proving liveness properties.

5.2 *L&S*

The formal proof method provides for structured proofs. Rules B1–B5 and C1–C8 explicitly decompose proofs of safety properties. Rules B6 and C9 and the proofs in the paper do not indicate how the proofs of liveness properties are structured. However, they involve standard temporal-logic reasoning, and can be decomposed using standard methods.

Although their method encourages rigorous, carefully structured proofs, Lam and Shankar do not present such a proof. Some things in the paper lead me to suspect that they have never carried a proof down to the arduous, boring level of detail I find necessary to avoid errors. For example, the version that I read omitted the assumption that NULL is not an element of the set VALUES. This assumption is so obvious that it is almost always taken for granted, and its omission not noticed, when writing journal-style proofs. But to someone who has written detailed, rigorous proofs, the need for this assumption is glaringly obvious upon reading the definition of *localvalue* in Section 5.1.

I do not mean to single out Lam and Shankar for their failure to provide an extremely rigorous proof. Such proofs are almost unheard of outside the mechanical verification community, and I have no reason to believe that either of the other authors have written them. It is only because their method is so conducive to rigorous proofs that the absence of one becomes obvious. It may turn out that the only way to get people to write rigorous proofs is to require machine checking.

5.3 *Br*

No proofs are given, so it is hard to judge what they would have looked like. Moreover, as discussed above, no implementation was specified, so one would even have to speculate about what was to be proved.

6 Completeness of the Proof Method

Completeness of a formal system means that every semantically valid assertion is provable. As Gödel showed, this is too much to hope for. For a state-based method, the relevant notion is completeness relative to the underlying formalism for reasoning about states, which means that one can deduce all valid formulas about specifications by assuming all valid formulas about states. Incompleteness of a method often indicates an important deficiency, indicating that the proof method is not strong enough.

Incompleteness of a method means that a particular valid formula cannot be proved. There may be a semantically (but not provably) equivalent formula that can be proved. Thus, even though it may be impossible to prove that a program implements a certain specification, it may be possible to prove that it implements an equivalent specification.

6.1 *K-S*

Since all reasoning in *K-S* is at the semantic level, it is not possible to talk about completeness of the method. However, we can discuss completeness of Back's refinement calculus, which provides a possible logical foundation for the method of *K-S*. Completeness of a state-based method for reasoning about execution sequences requires the ability to add two kinds of dummy variables (or their equivalent): history variables to record the past and prophecy variables to predict the future. Back's refinement calculus lacks prophecy variables, so it is incomplete. Moreover, this incompleteness is more than just a theoretical curiousity. The high-level specification of serializability in *K-S* requires an internal prediction of the future, which does not appear in any real implementation. Hence, a state-based verification that this specification is satisfied will require a prophecy variable.

There is another potential source of incompleteness in Back's calculus. In the process of encoding liveness properties in joint-action systems, Back creates programs that are not machine-closed—a technical property defined in [1]. The absence of machine closure can be a source of incompleteness for state-based methods. It is not known whether or not Back's encoding actually does introduce incompleteness.

6.2 *L&S*

This method also lacks anything equivalent to prophecy variables, so it is incomplete. The method is apparently based on the assumption that specifications will be written in such a way that prophecy variables are not needed to verify the correctness of implementations. While possible in principle, it is not clear how successful this will be in practice.

The highly constrained proof method of *L&S* results in another, somewhat surprising, source of incompleteness. Safety properties of an interface can be expressed either in the state transition part or in the invariant. It turns out that whether or not one can verify (using rules B1–B6 or C1–C9) that a module "offers" the interface may depend on whether the safety property is expressed in the state transition part or in the invariant.

6.3 *Br*

There is no formal separation between the state-based and the sequence-based part of a specification, so the concept of relative completeness is not relevant. The ability to reason directly about the execution history means that the sources of incompleteness in assertional formalisms should be easy to avoid. In fact, it seems possible to encode within the formalism of *Br* the reasoning used in assertional proofs. So, one can add axioms to provide at least the power of any desired assertional method, and there do exist relatively complete assertional methods.

7 Structure, Refinement, and Composition

Complicated specifications need some form of structure to be comprehensible. The specification must be broken into separate parts, which can be understood individually, that are then combined to understand the whole. There are two "dimensions" to this structuring, which might be called *refinement* and structuring *within a level*. Refinement changes the grain of atomicity, replacing one action by several. Structuring within a level preserves the grain of atomicity; it structures the description of a system's actions, but does not add new ones. Structuring within a level may be by *decomposing* a single system, or by *composing* open subsystems. Decomposition is carried out completely within the context of the given system; in composition, each subsystem is specified independently.

7.1 *K-S*

Breaking the system into individual actions is the primary form of structuring used in *K-S*. Additional structuring is provided by language mechanisms for extending previously defined objects. Structuring is by extending objects with new components and refining existing actions to describe how they change these components. This is decompositional. Composition is not discussed, except for the composition of the system and its environment. The joint action systems on which the method is based do permit composition, so composition should be possible. However, composing specifications that have liveness properties is a delicate matter, and cannot be taken for granted.

Refinement is provided by adding new actions and objects, in conjunction with the extension of existing objects. This type of refinement can be used to derive an implementation from a specification, which is most likely to occur when the specification is written as part of the process of designing a lower-level implementation.

There is a more general class of refinement that is not supported by the language—refinement in which the state structure is altered, not just extended with new components. Such a refinement is likely to be necessary when the higher-level specification is a generic one, such as an industry standard, not written with a particular implementation in mind. With such a refinement, establishing a relation between the higher- and lower-level specifications requires a "refinement mapping" [1] from lower-level objects to higher-level ones. As presented in *K-S*, the language does not include provision for such refinement mappings, so they can appear only at the semantic level when proving the correctness of the implementation.

7.2 *L&S*

Breaking a transition system into actions is the only method of structuring provided by the method. This is decompositional structuring. The method has no provision for composition. For example, one cannot show that the composition of two queues implements a larger queue, as in [6, Section 4.4]. One can only construct a module that "offers" a queue interface "using" two queue interfaces.

Refinement is provided by the alternating layers of interface specifications and module specifications. However, there are some annoying restrictions on the refinement process.

First, the restriction that an interface cannot have internal events will complicate some specifications. Nondeterministic external behavior is often easiest to describe in terms of the nondeterministic ordering of internal events. For example, the effect of concurrent write requests to the same memory location is easiest to describe in terms of the ordering of internal operations to the memory. With the method of *L&S*, a nondeterministic choice must be made at the time of the external requests.

Second, the generalization from the linear hierarchy to a tree hierarchy (mentioned but not described in *L&S*) requires that the lower-level interfaces be disjoint. Thus, it does not allow a situation in which one module is implemented using two lower-level interfaces, and the implementation of those lower-level interfaces shares a common interface—for example, a file system.

7.3 *Br*

Composition (not discussed in the paper) is through the simple and elegant mechanism of functional composition applied to the stream functions. Refinement could be achieved by applying some form of projection operator to a stream function. However, unlike methods in which sequences are generated by state/transition systems, *Br* allows arbitrary definitions of stream functions. This generality seems to make refinement less important.

Br offers no explicit method for decomposition. However, it contains a very convenient and powerful mechanism for decomposing a specification: mathematical definition. One can describe complex structures in small, simple steps through the proper use of definitions. On the other hand, their improper use can lead to an impenetrable maze of formalism. The mathematical elegance of the method should permit well structured specifications. I don't know if it encourages them in practice.

8 Practical Issues

I have found there to be two basic classes of methods for writing specifications. In the first class, a system is specified by a collection of properties, usually called axioms. I believe that such methods are impractical. It is very hard to understand the meaning of a list of abstract axioms, and I have seen intelligent computer scientists write incorrect axiomatic specifications of a simple FIFO queue. In the second class of methods, a system is specified by an abstract program.[3] Successful experiences with such methods are often attributed to the particular formalism or language that was used, when they are actually tributes to the power of the programming paradigm.

Although all programming languages are, in some sense, Turing complete, they differ in two important aspects: how elegant they are at programming "in the small" and how practical they are for writing large programs. The counterparts of these criteria in a specification language are: how simply can one express the individual components of a specification, and what support is provided for large specifications. Elegance in the small is more important for a specification language than for a programming language, since "coding hacks" to circumvent language deficiencies are less acceptable in a specification than in a program. Specification in the large is less of a problem than programming in the large, since specifications should be considerably smaller than programs.

8.1 *K-S*

Specifications are abstract programs, so the method should be practical. The language has been carefully designed to provide elegance in the small. However, a precise syntax for definitions has not been presented. Seemingly minor deficiencies, such as an insufficiently general type system, could cause serious problems in writing practical specifications. Although *K-S* provides a promising introduction to a language for specification in the small, it does not provide a complete description of one.

No consideration has been given to specification in the large. However, standard programming language concepts for modularizing programs should be adaptable to DisCo. The necessary enhancements to permit large specifications should be straightforward.

[3] The two classes of methods are sometimes called "axiomatic" and "operational", but those terms are misleading. With an axiomatic semantics for the programming language, any program is an axiomatic specification.

8.2 *L&S*

Here, module specifications are abstract programs, so they should be practical. An interface specifications is a combination of an abstract program (the state transition system) with invariance and progress axioms. My experience indicates that invariance axioms are fairly benign, but progress axioms can be quite tricky and can have unforeseen consequences. It is surprisingly easy to write an innocent-looking progress axiom that makes a specification unsatisfiable by any execution, so the progress axioms are a potential hazard for users. Unless practical guidelines for how to write progress axioms are developed, these axioms can easily lead to errors in complex specifications.

The relational notation for describing state transition systems is a good one, and is a definite plus for specification in the small. Other language issues, including the rules for writing the relations, are not considered. (For example, although other work by the authors suggests that they envision a strongly-typed language, this is not mentioned in *L&S*.) There is no mention of support for specification in the large. I believe that Lam and Shankar have concentrated their efforts on the basic foundation of their method, and have deferred consideration of such language issues.

8.3 *Br*

The fundamental basis of this method is that a specification is a collection of axioms. This would suggest that the method will be impractical. Indeed, Broy published a specification of a very simple elevator (lift) system, using this method, that was incorrect because it permitted undesired behavior [3]. (A corrected version was subsequently published [4].) Unrestricted use of the method leads to a style of specification that I believe is impractical for real systems.

However, the method admits a restricted style of specification-writing that essentially produces abstract programs. In this style, auxiliary functions play the role of program variables. Such a style should lead to practical specifications. It may also be possible safely to use some nonprogram-like axioms for the stream functions.

Br presents an extremely elegant approach to specification in the small. Issues of specification in the large are not treated. For example, practical specification will require the development of a library of data types. This will be difficult without some form of polymorphism, and the method's type system is not explained. Also, without some form of name hiding, naming conflicts can easily give rise to inconsistent specifications. However, since the algebraic approach has been extensively developed, I imagine that solutions to the problems posed by large specifications do exist.

9 Conclusion

Perhaps the most surprising result of the Lake Arrowhead workshop was how difficult the problem turned out to be. Lam and Shankar and Broy were the only speakers who made what I regarded to be serious attempts to address the problem. I am grateful to them and to Kurki-Suonio for their efforts, and I regret that others did not contribute to this issue. I have given my view of the three methods' strengths and weaknesses; the reader can draw his or her own conclusions. I hope that this exercise will make everyone working on formal methods better appreciate the difficulty of the task.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] R. J. R. Back. Refinement calculus, part ii: Parallel and reactive programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer-Verlag, May/June 1989.

[3] Manfred Broy. An example for the design of distributed systems in a formal setting: The lift problem. Technical Report MIP–8802, Fakultät für Mathematic und Informatik, Univerität Passau, 1988.

[4] Manfred Broy. Requirement and design specification for distributed systems: the lift problem. In *Proceedings of the Workshop on Future Trends of Distributed Computing Systems in the 1990s*, pages 164–173. IEEE Computer Society Press, 1988.

[5] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[6] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, London, 1985.

[7] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.