High-Level Specifications: Lessons from Industry

Brannon Batson
Intel Corporation

Leslie Lamport
Microsoft Research

11 Mar 2003

To appear in the Proceedings of the First International Symposium on Formal Methods for Components and Objects, held 5–8 November 2002 in Leiden, The Netherlands.

Abstract

We explain the rationale behind the design of the TLA$^+$ specification language, and we describe our experience using it and the TLC model checker in industrial applications—including the verification of multiprocessor memory designs at Intel. Based on this experience, we challenge some conventional wisdom about high-level specifications.

Contents

1 Introduction

The first author is a computer architect with a master's degree in electrical engineering. His work focuses on designing, implementing, and validating multiprocessor cache-coherence protocols. He has worked on TLA$^+$ formal specifications for the cache-coherence protocols of two Digital/Compaq Alpha multiprocessors, and he is currently using TLA$^+$ to model protocols on future Intel products.

The second author is a computer science researcher who began verifying concurrent algorithms over 25 years ago [12]. About ten years ago, he devised TLA, a logic for reasoning about concurrent algorithms [15]. He later designed TLA$^+$, a complete high-level specification language based on TLA [17].

The two authors view formal verification and TLA$^+$ from two different, complementary vantage points. In this paper, we try to synthesize our two views to explore the rationale behind TLA$^+$, describe our experience using it in industry, and derive some lessons from this experience. When discussing our individual experiences, we refer to the first and second authors as BB and LL, respectively.

We begin by describing TLA$^+$ and TLC, the TLA$^+$ model checker. We then describe how TLA$^+$ has been used at Digital/Compaq and at Intel. We next explore how our experience contradicts some conventional wisdom about specification, and we end with some simple conclusions.

2 TLA$^+$

2.1 Desiderata

TLA$^+$ is a high-level language for describing systems—especially asynchronous concurrent and distributed systems. It was designed to be simple, to be very expressive, and to permit a direct formalization of traditional assertional reasoning—the style of reasoning begun by Floyd [5] and Hoare [9] and extended to concurrent programs by Ashcroft [2], Owicki and Gries [21], Pnueli [24], and others [3, 11, 12, 22]. Making it easy, or even possible, to build tools was *not* a design criterion for the language.

The desire to formalize assertional reasoning, especially for liveness properties, led LL to base TLA$^+$ on TLA (the Temporal Logic of Actions) [15], a simple variant of linear-time temporal logic [24]. To be practical, a temporal logic must be based on an expressive language for writing elementary, non-temporal expressions. The desire for simplicity and expressiveness led to the use of ordinary first-order logic and set theory for this underlying language of expressions.

2.2 From Math to TLA$^+$

First-order logic and set theory provide a formalization of ordinary mathematics. TLA adds to them modalities for expressing temporal properties. Temporal modalities are useful for describing liveness (eventuality) properties. However, temporal logic, like any modal logic, is more complicated than ordinary math. TLA was therefore designed to put most of the complexity, both in describing a system and in reasoning about it, into the realm of ordinary math rather than into temporal logic.

LL originally assumed that first-order logic and set theory extended with the temporal operators of TLA would provide the semantic basis for a language, but that a practical language would require conventional programming-language constructs such as assignment statements. However, not wanting to introduce unnecessary constructs, he decided to begin writing formal specifications using only mathematics, and to add other constructs as needed. To his surprise, he discovered that he did not need those conventional constructs. Instead, he added to TLA$^+$ only the following extensions to ordinary mathematics:

Unambiguous Syntax A formal language must be unambiguous, meaning that it must be possible for a program to parse it. This led to eliminating from TLA$^+$ two common practices of mathematicians: using juxtaposition as an operator and the overloading of operators. Mathematicians write the product of x and y as xy; in TLA$^+$ it is written $x * y$. (One could require a space between x and y to distinguish this product from the single variable xy, but that would make parsing difficult.) Mathematicians frequently overload operators—for example, f^{-1} could mean either the inverse of f or f raised to the power -1. There is no overloading of operators in TLA$^+$. (The use of types can make some instances of overloading unambiguous; but for reasons explained below, TLA$^+$ is untyped.)

New Constructs TLA$^+$ borrows a few useful constructs from computer science—for example, allowing IF/THEN expressions like

$$\textbf{if } x \neq 0 \textbf{ then } 1/x \textbf{ else } 0$$

Also, mathematicians have no notation for explicitly describing a function—for example, the function whose domain is the set of reals and that maps every number to its negative. In TLA$^+$, this function is written

$$[x \in \mathit{Real} \mapsto -x]$$

(A computer scientist might write this as a λ expression, but TLA$^+$ avoids the keyword LAMBDA because of its potentially confusing connotations.)

Definitions Mathematicians have no standard convention for defining operators. They typically write something like "let $\circ$ be defined by letting $a \circ b$ equal ..., for any a and b." In TLA$^+$, one writes:

$$a \circ b \;\triangleq\; \ldots$$

Support for Large Specifications Mathematicians typically introduce new variables informally as needed. This casual introduction of variables could lead to errors in large specifications, so TLA$^+$ requires that variables be declared before they are used. Moreover, mathematicians will write "let x be an element of S" even though two pages earlier they had defined x to have some other meaning. Formalizing this requires some method of restricting the scope of a declaration. TLA$^+$ does this through the use of modules, which provide a mechanism for structuring large specifications. The mathematical operation of substituting expressions for variables is expressed in TLA$^+$ by instantiating a module with expressions substituted for its declared variables.

Support for Large Formulas For a mathematician, a 20-line formula is large. In a specification, a 200-line formula is not unusual. To aid in writing long formulas, TLA$^+$ allows bulleted lists of conjuncts and disjuncts, using indentation to eliminate parentheses [14]. For example,

$$\begin{array}{l} \wedge\ \mathit{reqQ}[p][i].\mathit{type} \neq \text{“}\mathsf{MB}\text{”} \\ \wedge\ \vee\ \mathit{DirOpInProgress}(p, \mathit{reqQ}[p][i].\mathit{adr}) \\ \quad\ \vee\ \mathit{reqQ}[p][j].\mathit{adr} \neq \mathit{reqQ}[p][i].\mathit{adr} \end{array}$$

means

$$\begin{array}{ll} & (\mathit{reqQ}[p][i].\mathit{type} \neq \text{“}\mathsf{MB}\text{”}) \\ \wedge & (\quad (\mathit{DirOpInProgress}(p, \mathit{reqQ}[p][i].\mathit{adr})) \\ & \ \ \vee\ (\mathit{reqQ}[p][j].\mathit{adr} \neq \mathit{reqQ}[p][i].\mathit{adr})) \end{array}$$

TLA$^+$ also has a LET/IN construct for making definitions local to an expression. This permits structuring an expression for easier reading as well as combining multiple instances of the same subexpression.

TLA TLA$^+$ extends ordinary math by adding the modal operators of TLA. The most important of these is prime ($'$), where priming an expression makes it refer to the value of the expression in the next state. For example, $x' = x + 1$ is an *action*, a predicate on a pair of states (called the current and next state), that is true iff the value of x in the next state is one greater than its value in the current state. Although formally a modal operator, expressions with primes obey the rules of ordinary mathematics, where x' is treated like a new variable unrelated to the variable x. TLA also has a few simple temporal operators, used mostly for expressing liveness properties. As we will see, these operators appear in only a small part of a specification.

We have listed the ways in which TLA$^+$ differs from math as used by ordinary mathematicians. Where practical, TLA$^+$ maintains the richness and economy of ordinary mathematical notation. For example, while a textbook on first-order logic typically defines only a simple quantified formula such as $\exists\, x : exp$, mathematicians typically write formulas like:

$$\exists\, x, y \in S, \langle z, w \rangle \in T : exp$$

TLA$^+$ allows this kind of richer syntax. On the other hand, mathematicians do not use extraneous keywords or punctuation. TLA$^+$ maintains this simplicity of syntax; for example, successive statements are not separated by punctuation. This syntactic economy makes TLA$^+$ specifications easy for people to read, but surprisingly hard for a program to parse.

2.3 A Brief Taste of TLA$^+$

To provide a sense of how a specification written in TLA$^+$ compares to one written in a typical specification language used in industry, Figures 1 and 2 give small parts of two versions of a toy specification. The first version is written in Promela, the input language of the Spin model checker [10]. (It was written by Gerard Holzmann, the designer of the language.) The second is written in TLA$^+$. The figures contain corresponding parts of the two specifications, although those parts are not completely equivalent. (One significant difference is mentioned in Section 3 below.)

Figure 3 shows a small part of a TLA$^+$ specification of a real cache-coherence protocol for a multiprocessor computer. Observe that it looks very much like the piece of toy specification of Figure 2; it is just a little more complicated. In both examples, the TLA$^+$ specification uses only simple mathematics.

```
inline AppendNum(n)
 { i = 0;
   do :: i < MaxSeqLen
      && seq[i] != 0
      && seq[i] != n -> i++
      :: else -> break
   od;
   if :: i >= MaxSeqLen
      || seq[i] != 0
      :: else -> seq[i] = n
   fi  }
```

Figure 1: Part of a toy specification written in Promela.

$$
\begin{array}{l}
AppendNum(n) \;\triangleq \\
\quad \land\; \forall\, i \in 1 \,..\, Len(seq) \;:\; n \neq seq[i] \\
\quad \land\; seq' = Append(seq, n) \\
\quad \land\; num' = num
\end{array}
$$

Figure 2: The TLA$^+$ version of the piece of specification in Figure 1.

$$
\begin{array}{l}
\land\; req.type = \text{“MB”} \\
\land\; \forall\, i \in 1 \,..\, (idx - 1) \;: \\
\qquad \land\; reqQ[p][i].type \neq \text{“MB”} \\
\qquad \land\; DirOpInProgress(p, reqQ[p][i].adr) \\
\qquad \land\; \forall\, j \in 1 \,..\, (i - 1) \;:\; reqQ[p][j].adr \neq reqQ[p][i].adr \\
\land\; \neg\, \exists\, m \in msgsInTransit \;: \\
\qquad\quad \land\; m.type \in \{\text{“Comsig”}, \text{“GetShared”}, \text{“GetExclusive”}, \\
\qquad\qquad\qquad\qquad \text{“ChangeToExclusive”}\} \\
\qquad\quad \land\; m.cmdr = p
\end{array}
$$

Figure 3: A small piece of a real TLA$^+$ specification.

Crucial to the simplicity of TLA$^+$ is that it is based on the ordinary mathematics used by ordinary mathematicians. Computer scientists have devised many forms of weird mathematics. They have introduced bizarre concepts such as "nondeterministic functions", leading to strange formalisms in which the formula $A = A$ is not necessarily true. Ordinary mathematics was formalized about a century ago in terms of first-order logic and (untyped) set theory. This is the formal mathematics on which TLA$^+$ is based. The use of ordinary mathematics in TLA$^+$ led BB to remark: if I want to find out what an expression in a TLA$^+$ specification means, I can just look it up in a math book.

Computer scientists and engineers, accustomed to computer languages, are likely to be surprised by the expressiveness of TLA$^+$. BB describes the power of TLA$^+$ in this way:

> A single line of TLA$^+$ can do powerful manipulations of complex data structures. This allows me to focus on the algorithm without getting lost in bookkeeping tasks. TLA$^+$ tops even perl in this regard. Unlike perl, however, TLA$^+$ is unambiguous.

The simplicity and expressiveness of TLA$^+$ is not the result of any cleverness in the design of the language; it comes from two thousand years of mathematical development, as formalized by great mathematicians like Hilbert. TLA$^+$ is described in a recent book by LL, which is available on the Web [17].

2.4 The Structure of a TLA$^+$ Specification

TLA$^+$ does not enforce any particular way of structuring a specification, allowing declarations and definitions to appear in any order as long as every identifier is declared or defined before it is used. Moreover, by partitioning it into separate modules, one can structure a specification to allow reading higher-level definitions before reading the lower-level definitions on which they depend.

Logically, most TLA$^+$ specifications consist of the following sections:

Declarations The declarations of the constant parameters and the variables that describe the system. There are typically a dozen or so declared identifiers.

Definitions of Operations on Data Structures These define mathematical operators used to describe operations specific to the particular system. For example, a specification of a system that can perform a masked store to a register might define

$$MaskedStoreResult(curr, val, msk)$$

to be the new value of a register, whose current value is *curr*, after storing to it a value *val* through a mask *msk*. A specification usually has only a few such operator definitions, each just a few lines long. The operator *MaskedStoreResult* is more likely to be used only in the definition that describes the masked store action, in which case it would probably be defined locally in a LET/IN expression.

The Initial Predicate The definition of a formula that describes the possible initial values of the variables. It is a conjunction of formulas $x = \ldots$ or $x \in \ldots$ for each variable x, where the "..." is usually a simple constant expression.

The Next-State Action The definition of a formula containing primed and unprimed variables that describes the system's possible next state as a function of its current state. It is generally defined as a disjunction of subactions, each describing one type of system step. For example, in the specification of a mutual-exclusion algorithm, the next-state action might have a disjunct $\exists\, p \in Proc : EnterCS(p)$, where $EnterCS(p)$ describes a step in which a process p enters its critical section. The definition of the next-state action comprises the bulk of the specification.

Liveness The definition of a temporal formula specifying the liveness properties of the system, usually in terms of fairness conditions on subactions of the next-state action. It typically consists of about a dozen lines.

The Specification This is the one-line definition

$$Spec \triangleq Init \wedge \Box[Next]_v \wedge Liveness$$

that defines *Spec* to be the actual specification, where *Init* is the initial predicate, *Next* is the next-state action, *Liveness* is the liveness formula, and v is the tuple of all variables.

For a high-level specification that describes a system's correctness properties, *Spec* is the *inner* specification in which internal variables are visible. The true specification would be obtained by hiding those internal variables. If h is the tuple of all internal variables, then

Spec with those variables hidden is represented by the TLA formula $\exists\!\!\!\exists\, h : \mathit{Spec}$. For technical reasons, TLA$^+$ requires that this formula be defined in a separate module from the one defining *Spec*. However, hiding the internal variables is done for philosophical correctness only. The TLC model checker cannot handle the TLA hiding operator $\exists\!\!\!\exists$, and in practice, one uses the inner specification *Spec*.

The only temporal operators that appear in the entire specification are the ones in the liveness formula and the single $\Box$ in the final specification. The rest of the specification—usually about 99% of it—consists entirely of ordinary math, with no temporal operators. Moreover, engineers generally do not use the liveness property. The complexity of model checking liveness properties is inherently much greater than that of checking safety properties, which means that liveness can be checked only for extremely small models of real systems. Engineers therefore usually do not even write the liveness property; instead their specification is

$$\mathit{Spec} \;\triangleq\; \mathit{Init} \wedge \Box[\mathit{Next}]_v$$

The only temporal operator in this specification is the single $\Box$.

2.5 The Size of Specifications

In principle, one could write specifications of any size, from tiny toy examples to million-line monsters. But tiny toys are of no practical interest, and the module structure of TLA$^+$ is probably inadequate for handling the complexity of specifications longer than ten or twenty thousand lines. We have identified the following three classes of applications for which TLA$^+$ is useful, each with a surprisingly narrow range of specification sizes:

Abstract Algorithms These are the types of concurrent algorithms that are published in journals—for example, the Disk Paxos algorithm [6]. Their specifications seem to require a few hundred lines of TLA$^+$. Interesting algorithms simple enough to have much shorter specifications seem to be rare, while an algorithm with a much longer specification is probably not abstract enough for journal publication.

Correctness Properties These are descriptions of the properties that protocols or systems should satisfy. One example is a description of the memory model that a cache-coherence protocol is supposed to implement [20]. Their specifications also seem to require a few hundred lines of TLA$^+$. The requirements of a real system are seldom simple enough to have a shorter specification, while a statement of correctness requiring a much longer specification would probably be too complicated to be useful.

High-Level Protocol or System Designs These describe the high-level designs of actual systems or protocols. We know of no published example; such designs are usually proprietary. We have found that these specifications are about two thousand lines of TLA$^+$. Any such specification is an abstraction of the actual lower-level implementation. Engineers want to describe their design in as much detail as they can. However, if the specification takes much more than two thousand lines, then the design is too complicated to understand in its entirety, and a higher-level abstraction is needed.

3 TLC: The TLA$^+$ Model Checker

TLA$^+$ was not designed with tools in mind; LL believed that a practical model checker for it was impossible and advised against trying to write one. Fortunately, Yuan Yu ignored him and wrote TLC, an explicit-state model checker for TLA$^+$ programmed in Java [26].

TLA$^+$ is an extremely expressive language—for example, it can easily be used to specify a program that accepts an arbitrary Turing machine as input and tells whether or not it will halt. No model checker can handle all TLA$^+$ specifications. TLC handles a subset of TLA$^+$ that seems to include most specifications of algorithms and correctness properties, as well as all the specifications of protocol and system designs that engineers actually write. Those few specifications arising in practice that TLC does not handle can be easily modified, usually by changing only a few lines, so they can be checked by TLC.

Explicit-state model checking is possible only for bounded-state specifications. Most high-level specifications are not bounded-state because the state contains data structures such as unbounded queues. We want engineers to use the TLA$^+$ specification as the official high-level specification of their system, and a major goal of TLC is that the specification should not have to be changed to allow it to be checked. So TLC accepts as input a TLA$^+$ specification and a configuration file that defines a finite model. The configuration file instantiates the constant parameters of the specification—for example, instructing TLC to replace the parameter *Proc* that represents the set of processors with a set containing three elements. The configuration file can also specify a constraint on the state space, instructing TLC to explore only states satisfying the constraint.

As an example, we return to the toy specification, part of which is specified in Figures 1 and 2. In that specification, the variable *seq* represents a queue that can grow arbitrarily large. To model check it with TLC, we write a simple module that imports the original specification, declares a constant *MaxSeqLen*, and defines a constraint asserting that the length of *seq* is at most *MaxSeqLen*. We then instruct TLC to check the specification using that constraint, substituting a specific value for *MaxSeqLen*. We do this for increasing values of *MaxSeqLen* until we are confident enough that the specification is correct, or until the space of reachable states becomes so large that it takes TLC too long to explore it. In contrast, note in Figure 1 that, to model check the specification with Spin, the parameter `MaxSeqLen` had to be made part of the actual Promela specification.

Operations on many common data types are not built into TLA$^+$, but are instead defined in standard modules. For example, the natural numbers are defined in the *Naturals* module to be an arbitrary set satisfying Peano's axioms, and arithmetic operation are defined in the usual way in terms of the next-number function. A specification that uses operators like $+$ on natural numbers imports the *Naturals* module. It would be rather difficult for TLC to compute $2 + 2$ from the definition of $+$ in that module. Instead, such arithmetic operations are programmed in Java using TLC's general module overriding mechanism. When a specification imports a module named M, TLC looks for a Java class file named `M.class`. If it finds one, it replaces operators defined in M with their Java implementations in `M.class`. There are Java implementations for common operators on numbers, sequences (lists), finite sets, and bags (multisets) that are defined in the standard modules. Ordinary users can also write their own Java class files to provide more efficient implementations of the operators that they define. However, we know of no case where this was necessary, and we know of only one user (a researcher) who has done it.

TLC has a primitive command-line interface. Debugging is done by adding print statements to the specification. Although this violates the principle of not having to modify the specification to check it, the print statements are usually removed as soon as initial debugging is completed and simple "coding" errors corrected. Design errors are generally found by examining error traces. We hope eventually to add a graphical user interface.

TLC is coded in Java. It is a multithreaded program and there is a version that can use multiple machines. For checking safety properties, it obtains close to an n-fold speedup with n processors when run on Alphas using a high quality Java runtime. However, we have found that the poor implementation of multithreading in many Java runtimes can significantly reduce the speedup. The largest case we know of was one with 900 million reachable states that took about two weeks on a four-processor machine.

The expressiveness of TLA$^+$ makes it essentially impossible to compile TLA$^+$ specifications into efficient code. Therefore, TLC must interpret specifications. We guess that this makes TLC about ten times slower than explicit-state model checkers that require specifications to be written in a low-level, compilable language. Because TLC maintains its data structures on disk, it has essentially no space limitations for checking safety properties.

The goal of TLC is to help engineers find bugs in their designs. Experience tells an engineer what kind of bugs a particular finite model is and is not likely to find. For example, if a cache-coherence protocol handles different addresses independently, then it may suffice to check models with only a single memory address. A specification is an abstraction, and checking it can find only errors that are present in that abstraction. Lower levels of detail introduce many other sources of error not reflected in the specification. In most industrial applications, engineers need cost-effective methods of finding bugs; they are not seeking perfection.

4 TLA$^+$ Use in Industry

4.1 Digital/Compaq/HP

TLA$^+$ and TLC were conceived at the Digital (later Compaq) Systems Research Center. The first serious industrial use of TLA$^+$ was for specifying and writing part of a hand proof of the cache-coherence protocol of a multiprocessor code-named Wildfire, based on the Alpha EV6 processor [7].

The Wildfire experience inspired Yuan Yu to write TLC. It also persuaded the verification team to write a TLA$^+$ specification of the cache-coherence protocol of the next generation Alpha, the EV7. The initial specification viewed an entire processor chip as a single component; the level of abstraction was later lowered to model protocol interactions between on-chip components as well. TLC checked important properties of the protocol and helped find several bugs. TLC and the EV7 protocol specification were also used as the basis of a project to improve test coverage for hardware simulation [25].

The processor design team for the next Alpha processor, the EV8, began using TLA$^+$ to write the official specification of its cache-coherence protocol. However, development of that processor was cancelled.

TLA$^+$ and TLC were also applied by Compaq engineers to the cache-coherence protocols of two Itanium-based processors. Researchers used TLC to debug a bus protocol proposal and to help develop database recovery and cache management protocols. TLC is now used routinely by some researchers to check the concurrent algorithms that they develop.

The use of TLA$^+$ and TLC at Digital and Compaq, some of which continued at HP, is described in [18].

4.2 Intel

We now describe the use of TLA$^+$ and TLC by BB and his colleagues at Intel. The actual specifications are proprietary and have not been viewed by anyone outside Intel.

4.2.1 Overview of the Problem

Designing a complex system starts with a problem statement and an appropriate set of boundary conditions. A component of a computer system is initially represented abstractly as a black box, with assertions about its functionality and with some guidelines on performance, cost, and complexity. The engineering process involves iteratively refining this abstract model into lower-level models. Each lower-level model is a representation of the design at a certain level of abstraction, and it has a specific purpose. Some models are meant to evaluate tradeoffs between scope, performance, cost, and complexity. Others carry the design down to the low level of detail needed to manufacture the component.

The engineering process therefore creates multiple representations of a design. Validation entails checking these multiple representations against one another. Designers of digital systems have good tools and methods for validating mid-level functional models, written in a hardware description language (HDL) like VHDL or RTL, against lower-level models such as circuit net-lists. However, they have not had as much success checking the higher-level functional representations of the design against one another, and against the initial problem statement and functional assertions.

For some components, there is an intuitive correlation between the high-level notions of correctness and the HDL model; such components tend not to be difficult to validate. Other components, like multiprocessor cache-coherence protocols, are sufficiently complex that checking the HDL model against the problem statement is quite challenging. We need formal techniques from the world of mathematics to perform this high-level validation.

Although formal methods are based on mathematics, engineers view them differently from the way mathematicians do. To engineers, formal verification is simply another imperfect validation tool (albeit a powerful one). A TLA$^+$ specification is only an abstraction of the actual system, and model checking can usually validate the specification only for a highly restricted set of system parameters. Validating the specification therefore cannot guarantee that there are no errors in the system. For engineers, formal verification is a way of finding bugs, not of proving correctness.

The main benefit of applying TLA$^+$ to engineering problems comes from the efficiency of the TLC model checker in reaching high levels of coverage and finding bugs. A secondary benefit we have encountered is the ability of TLA$^+$ and TLC to provide good metrics for the complexity of a design. Complexity is a major consideration in evaluating design tradeoffs. However, unlike performance or cost, engineers have not historically had a good way to quantify algorithmic complexity before attempting to validate a design. TLA$^+$ encourages designers to specify the design abstractly, suppressing lower-level details, so the length of the specification provides a measure of a design's complexity. TLC reports the size of the reachable state space, providing another measure of complexity. Experience and intuition will always have a place in evaluating complexity, but TLA$^+$ and TLC provide robust and impartial input to the evaluation. Having this input early in the design process is of considerable value.

4.2.2 Designing with TLA$^+$

The core group at Intel started using TLA$^+$ at Compaq while working on the Alpha EV7 and EV8 multiprocessor projects described above. From that experience, the Alpha engineers learned that multiprocessor cache-coherence protocols are an ideal candidate for formal methods because most of the protocol bugs can be found at a high level of abstraction. They also learned that the true value of TLA$^+$ and TLC would be realized when (a) they were applied early enough in the design to provide implementation feedback, and (b) the implementation was based directly on the specification that had been verified. On the EV8 project, the TLA$^+$ specification was completed before the design was stable, and it provided important feedback to the designers.

When the engineers from the Alpha group joined Intel, they began applying their experience in writing TLA$^+$ specifications when collaborating with other Intel engineers on cache-coherence protocols for future Intel products. Intel engineers are now using TLA$^+$ as an integral part of the design process for the protocols that are under study.

Whiteboard Phase Designing one cache-coherence protocol from scratch provided the engineers with the opportunity to evaluate TLA$^+$ as a prototyping platform for complex algorithms. Work on this protocol started by exploring the design space on a whiteboard for about two months. In this phase, basic message sequencing was determined, as were some coarse notions of what state had to be recorded at the protocol endpoints. A basic direction was set, based on the guidelines for functionality, performance, and cost.

Because of their background, engineers tend to visualize an algorithm in terms of a particular implementation. They are better at gauging implementation complexity than at measuring algorithmic complexity. One benefit of having engineers write formal specifications is that it helps them learn how to think about a protocol abstractly, independent of implementation details. We found that, even in the whiteboard phase of the protocol design, the Intel engineers were able to make some judgments on complexity by asking themselves, "How would I code this in TLA$^+$?".

The whiteboard phase produced a general understanding of the protocol philosophy, an understanding of the constraints placed on the communication medium, the basic message flows, and coarse ideas on what state needed to be maintained. The next step was to introduce the rigor of a formal specification.

TLA$^+$ Scratchpad Phase The TLA$^+$ scratchpad phase of the project involved formally describing the abstract system, with appropriate state variables representing high-level components. This phase took about two months, starting with the initial design of the protocol. The difficulty lay not in the use of TLA$^+$— engineers frequently learn new programming languages—but rather in (a) determining the layer of abstraction and (b) exploring the protocol's corner cases. Task (a) is where TLA$^+$ forces engineers to think about the protocol abstractly, which they often find unnatural. Their ability to think abstractly improves with experience writing TLA$^+$ specifications. Task (b) is inevitable when documenting a protocol formally, as it forces the designers to explore the corner cases. During the scratchpad phase, the designers had to return to the whiteboard a few times when they encountered new race cases while writing the specification.

The actions that formed the major blocks of the specification were chosen early; very few changes were made later. The Intel engineers adopted a methodology used in the earlier Alpha specifications, in which the decomposition of high-level named actions is based on classifying the protocol messages that they process. This methodology has led to fairly readable specifications, since it means that each action changes only a few local state variables. It encouraged the protocol specifications to be designed in a modular way, which also enabled the inter-module interfaces in the specification to be similar to their low-level counterparts in the implementation.

Running TLC The initial week or so of running TLC was spent finding and fixing typographical errors and type mismatch problems. This time could probably have been shortened by doing more syntax checking when writing the specification, which is what one often does when programming.

The next four weeks saw a continuous process of running TLC, finding bugs, fixing them, and re-running TLC. During this phase of the project, many assumptions and assertions about the protocol were brought into question. This had the effect of educating the engineers about the protocol they had designed. We have found that TLC can be a useful learning tool if we use in-line assertions and global invariants to check everything we think is true. The Intel engineers were able to develop an intuitive understanding of the correctness of the protocol by developing meaningful global invariants and having TLC check them. If an assertion or invariant fails, TLC generates a counterexample that is useful for visualizing a difficult race case. These counterexamples are such a powerful teaching aid that the Intel engineers have developed tools to translate the TLC output into nicely formatted protocol flow diagrams that are easier to read.

Another useful feature of the TLC model checker is its coverage checking. TLC can print the number of times each action was "executed". This provides a simple way to identify holes in coverage. Much of the effort expended by the engineers in debugging the specification was spent eliminating each of these holes, or convincing themselves that it represented an action that could never happen.

The performance of the model checker was sufficient to debug a large protocol specification. The engineers determined a base configuration that would "execute" all the actions and that displayed all interesting known cases. This configuration could be run on a four-processor machine in about a day, enabling fast turn-around on bug fixes. Larger configurations were periodically run as sanity checks on the smaller ones. The engineers would also run TLC in simulation mode, which randomly and non-exhaustively explores the state space, allowing them to check much larger configurations. Such random simulations are similar to the ones engineers typically perform on lower-level models, but it has the advantage of being several orders of magnitude faster because it is based on the abstract TLA$^+$ model, and it provides a robust metric for coverage.

4.2.3 Optimizing with TLC

Once the initial protocol specification was successfully checked by TLC, the Intel engineers were able to use it as a test bed for exploring optimizations. TLA$^+$ is an ideal language to explore changes because its expressiveness usually allows the new version to be written quickly. Model checking the modified specification with TLC not only checks functional correctness, but it also measures any increase in the state space. Such an increase implies additional algorithmic complexity. The engineers spent several months exploring additions to the protocol, testing them with TLC. As a general rule, they would consider adopting only those optimizations that did not appreciably expand the state space. The insight that TLA$^+$ and TLC gave into the complexity of modifications to the protocol was invaluable in iterating towards an optimal solution that adequately weighed algorithmic complexity along with factors like cost and performance.

A significant optimization was later made to the protocol. This optimization followed the normal design cycle described above, though on a compressed schedule. With the original design yielding a good starting point, the entire cycle (whiteboard phase, TLA$^+$ coding, and verification with TLC) was done within six weeks. This modification was accomplished by a recent college graduate with an undergraduate degree in engineering. He was able to learn TLA$^+$ well enough within a matter of weeks to do this work.

4.2.4 Feedback on TLA$^+$ syntax

The feedback we have received from engineers about the TLA$^+$ language has been mostly positive. Engineers are usually able to pick up and understand a specification within a couple of days. One mistake we made was to present TLA$^+$ to hardware designers as similar to a programming language. This led to some frustration. A better approach seems to be to describe TLA$^+$ as being like a hardware description language. Engineers who design digital systems are well acquainted with methods for specifying finite-state machines, with the associated restrictions of allowing a primed variable to be assigned a value only once within a conjunction, not allowing a primed variable to appear in a conjunction before the assignment of its value, etc. To an engineer, TLA$^+$ looks like a language for specifying finite-state machines.

While writing the protocol specification at Intel, BB was impressed by the ease of specifying complex data structures in TLA$^+$ as sets and tuples. The part of the specification that described and manipulated data structures was a small part of the complete protocol specification. This compact specification of "bookkeeping tasks", along with the overall expressiveness of TLA$^+$, won over the engineers who were accustomed to using more clumsy functional languages for specifying complex algorithms.

For the algorithmic specification, TLA$^+$ naturally encourages nested disjunctions of conjunctions (known to engineers as sums of products of expressions). This method for specifying Boolean formulas has both advantages and disadvantages. One advantage is that it allows expressive comment blocks and assertions to be inserted in-line with a nested conjunct. A disadvantage is that this tends to lead to large specifications. The engineers are experimenting with the use of TLA$^+$ operators to encode large blocks of regular Boolean disjunctions as truth tables, which engineers find more natural to work with.

5 Some Common Wisdom Examined

Based on our experience using TLA$^+$, we now examine the following popular concepts from the world of programming: types, information hiding, object-oriented languages, component-based/compositional specifications, hierarchical description/decomposition, and hierarchical verification. Most of these concepts were mentioned in this symposium's call for papers.

We do not question the usefulness of these concepts for writing programs. But high-level specifications are not programs. We find that in the realm of high-level specifications, these ideas are not as wonderful as they appear.

5.1 Types

Very simple type systems are very restrictive. Anyone who has programmed in Pascal has written programs that were obviously type-correct, but which were not allowed by Pascal's simple type system.

Moderately complicated type systems are moderately restrictive. A popular type system is that of higher-order logic [8]. However, it does not allow subtyping. With such a type system, an integer cannot be a real number. When writing Fortran programs, one gets used to 1.0 being unequal to 1. One should not have to put up with that kind of complication in a specification language.

Subtyping is provided by predicate subtyping, perhaps best known through its use in the PVS verification system [23]. We will see below a problem with PVS's predicate subtyping. Moreover, predicate subtyping is not simple. It has led to several bugs that caused PVS to be unsound.

For a typed language to be as expressive as TLA$^+$, it will need an extremely complicated type system, such as that of Nuprl [4]. Engineers have a hard enough task dealing with the complexity of the systems that they design; they don't want to have to master a complicated type system too.

A specification consists of a large number of definitions, including many local ones in LET/IN expressions. Although an operator may have a simple type, it is often hard or impossible to declare the types of the operators defined locally within its definition. Even when those type declarations are possible, LL has found that they clutter a specification and make it harder to read. (Early precursors of TLA$^+$ did include type declarations.) Any information contained in a type declaration that is helpful to the reader can be put in a comment.

The main virtue of types, which makes us happy to bear the inconvenience they cause when writing programs, is that they catch errors automatically. (That advantage disappears in type systems with predicate subtyping, in which type checking can require manually guided theorem proving.) However, we have found that the errors in a TLA$^+$ specification that could have been found by type checking are generally caught quite quickly by running TLC with very small models.

The problems with types are discussed at length in [19]. We want to emphasize that we do not dispute the usefulness of types in programming languages. We prefer to program in strongly typed languages. We are questioning the use of types only in a high-level specification language.

5.2 Information Hiding

We have learned that programmers should hide irrelevant implementation details. However, a high-level specification should not contain implementation details. Such details will appear in a well-written specification only if an inexpressive language requires high-level concepts to be described by low-level implementations. TLA$^+$ provides users with powerful mathematical objects like sets and functions; they don't have to encode them in arrays of bits and bytes. Such "bookkeeping details" do not occur in specifications written in a truly high-level language like TLA$^+$, so there is no need to hide them.

5.3 Object-Oriented Languages

The mathematical concept underlying object oriented programming languages can be described as follows. A program maintains identifiers of (references to) objects. There is a function *Obj* that maps the set *ObjectId* of object identifiers to a set *Object* of objects. An object-oriented language simply hides the function *Obj*, allowing the programmer to write *o.field* instead of $Obj[o].field$, where o is an object identifier.

Eliminating explicit mention of *Obj* can make a specification look a little simpler. But it can also make it hard to express some things. For example, suppose we want to assert that a property $P(obj)$ holds for every object *obj*. (Usually, $P(obj)$ will assert that, if *obj* is a non-null object of a certain type, then it satisfies some property.) This is naturally expressed by the formula

$$\forall\, o \in ObjectId \;:\; P(Obj[o])$$

It can be difficult or impossible to express in a language that hides *Obj*.

Object-orientation introduces complexity. It raises the problem of aliasing. It leads to the confusing difference between equality of object identifiers and equality of objects—the difference between `o1 = o2` and `o1.equals(o2)`. You can't find out what `o1.equals(o2)` means by looking it up in a math book.

Object-oriented programming languages were developed for writing large programs. They are not helpful for two-thousand-line programs. Object orientation is not helpful for two-thousand-line specifications.

5.4 Component-Based/Compositional Specifications

A high-level specification describes how the entire system works. In a TLA$^+$ specification, a component is represented by a subexpression of the next-state relation—usually by a disjunct. We can't understand a formula by studying its subexpressions in isolation. And we can't understand a system by studying its components in isolation. We have known for 20 years that the way to reason about a distributed system is in terms of a global invariant, not by analyzing each component separately [13].

Many tools have been developed for debugging the low-level designs of individual hardware components. Engineers need a high-level specification to catch bugs that can't be found by looking at individual components.

5.5 Hierarchical Description/Decomposition

Hierarchical description or decomposition means specifying a system in terms of its pieces, specifying each of those pieces in terms of lower-level pieces, and so on. Mathematics provides a very simple, powerful mechanism for doing this: the definition. For example, one might define A by

$$A \triangleq B \lor C \lor D$$

and then define B, C, and D. (TLA$^+$ requires that the definitions appear in the opposite order, but one can present them in a top-down order by splitting the specification into modules.)

Building up a specification by a hierarchy of definitions seems simple enough. But a specification language can make it difficult in at least two ways:

- It can restrict the kinds of pieces into which a definition can be broken. For example, it might require the pieces to be separate processes. There is no reason to expect that splitting the system into separate processes will be the best way to describe it.

- It can use a strict type system. For example, suppose x is a variable of type real number, and we want to define an expression A by

  $$A \triangleq \textbf{if } x \neq 0 \textbf{ then } B \textbf{ else } C$$

  where B is defined by

  $$B \triangleq 1/x$$

  This is a perfectly reasonable definition, but PVS's type system forbids it. PVS allows the expression $1/x$ only in a context in which x is different from 0. This particular example is contrived, but TLA$^+$ specifications often contain local definitions in LET/IN expressions that are type-correct only in the context in which they are used, not in the context in which they are defined.

How Intel engineers use and don't use hierarchical decomposition is somewhat surprising. As we observed above, the major part of a TLA$^+$ specification is the definition of the next-state action. Intel engineers use the common approach of decomposing this definition as a disjunction such as

$$Next \triangleq \exists\, p \in Proc : A_1(p) \lor \ldots \lor A_n(p)$$

where each $A_i(p)$ describes a particular operation performed by process p. They also use local definitions to make a definition easier to read. For example, an action $A_i(p)$ might be defined to equal

$$\begin{array}{ll} \text{LET} & newV \ = \ \ldots \\ & newW = \ \ldots \\ \text{IN} & \ldots \\ & \wedge\ v' \ = newV \\ & \wedge\ w' = newW \\ & \ldots \end{array}$$

This allows a reader to scan the IN clause to see what variables the action changes, and then read the complex definitions of $newV$ and $newW$ to see what the new values of v and w are.

What the Intel engineers do not do is use hierarchical decomposition to hide complexity. For example, they would not eliminate common subexpressions by writing

$$\begin{array}{lcl} SubAction(p) & \triangleq & \ldots \\ A_1(p) & \triangleq & \ldots \wedge SubAction(p) \wedge \ldots \\ A_2(p) & \triangleq & \ldots \wedge SubAction(p) \wedge \ldots \end{array}$$

if the two instances of $SubAction(p)$ represent physically distinct components.

The Intel engineers rely on the TLA$^+$ specification to gauge the complexity of their designs, using the number of lines in the specification as a measure of a design's complexity. This is possible because TLA$^+$ does not introduce the extraneous details needed by lower-level languages to encode higher-level concepts.

5.6 Hierarchical Verification

Hierarchical verification works as follows: To show that a system described by the specification *Sys* implements the correctness properties *Spec*, we write an intermediate-level spec *Mid* and show that *Sys* implements *Mid* and *Mid* implements *Spec*. In TLA$^+$, implementation is implication. To show that a specification $\exists\, x : F$ implements a specification $\exists\, y : G$, we must show

$$(\exists\, x : F) \Rightarrow (\exists\, y : G)$$

Here, x and y are tuples of variables, which we assume for simplicity to be distinct. By simple logic, we show this by showing

$$F \Rightarrow (G \;\textbf{with}\;\; y \leftarrow \mathit{exp})$$

for some tuple of expressions exp, where **with** denotes substitution (which is expressed in TLA$^+$ by module instantiation). The tuple exp is called a *refinement mapping* [1].

To show that *Sys* implies *Spec*, we must show

$$\mathit{Sys} \;\Rightarrow\; (\mathit{ISpec} \;\textbf{with}\;\; h \leftarrow \mathit{exp})$$

where *ISpec* is the inner specification, with internal variables h visible. To use hierarchical verification, we find an intermediate-level inner specification *IMid*, with internal variables m visible, and show:

$$\begin{aligned} \mathit{Sys} &\;\Rightarrow\; (\mathit{IMid} \;\textbf{with}\;\; m \leftarrow \mathit{exp}1) \\ \mathit{IMid} &\;\Rightarrow\; (\mathit{ISpec} \;\textbf{with}\;\; h \leftarrow \mathit{exp}2) \end{aligned}$$

When the verification is done by TLC, there is no reason for such a decomposition; TLC can verify directly that *Sys* implements *ISpec* under the refinement mapping. When the verification is done by mathematical proof, this decomposition seems reasonable. However, as LL has argued elsewhere [16], it is just one way to decompose the proof; it is not necessarily the best way.

Unfortunately, the whole problem of verifying that a high-level design meets its specification is not yet one that is being addressed in the hardware community. Thus far, engineers are checking only that their TLA$^+$ design specifications satisfy an incomplete set of invariants. Checking that they satisfy a complete specification is the next step. Engineers want to do it, but they have to learn how—which means learning how to find a correct refinement mapping. TLC has the requisite functionality to do the checking, but only doing it for real systems will tell if it works in practice. The reader can get a feeling for the nature of the task by trying to solve the Wildfire Challenge Problem [20].

6 Conclusions

Our industrial experience with TLA$^+$ has led us to some simple, common-sense conclusions:

- Buzzwords like *hierarchical* and *object-oriented* are to be viewed with suspicion.

- A language for writing high-level specifications should be simple and have effective debugging tools.

- Only proofs and model checking can catch concurrency bugs in systems. For the vast majority of applications, proofs are not a practical option; engineers have neither the training nor the time to write them.

- A specification method cannot be deemed a success until engineers are using it by themselves.

TLA$^+$ and TLC are practical tools for catching errors in concurrent systems. They can be used very early in the design phase to catch bugs when it is relatively easy and cheap to fix them. Writing a formal specification of a design also catches conceptual errors and omissions that might otherwise not become evident until the implementation phase.

TLA$^+$ is not just for industrial use. Anyone who writes concurrent or distributed algorithms can use it. We invite the reader to give it a try.

References

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

[2] E. A. Ashcroft and Z. Manna. Formalization of properties of parallel programs. In *Machine Intelligence*, volume 6. Edinburgh University Press, 1970.

[3] K. Mani Chandy and Jayadev Misra. *Parallel Program Design.* Addison-Wesley, Reading, Massachusetts, 1988.

[4] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panagaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[5] R. W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Math., Vol. 19*, pages 19–32. American Mathematical Society, 1967.

[6] Eli Gafni and Leslie Lamport. Disk paxos. To appear in *Distributed Computing.*, 2002.

[7] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Van Doren. Architecture and design of AlphaServer GS320. In Anoop Gupta, editor, *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 13–24, November 2000.

[8] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

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

[10] Gerard Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[11] Simon S. Lam and A. Udaya Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4):325–342, July 1984.

[12] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[13] Leslie Lamport. An assertional correctness proof of a distributed algorithm. *Science of Computer Programming*, 2(3):175–206, December 1982.

[14] Leslie Lamport. How to write a long formula. *Formal Aspects of Computing*, 6:580–584, 1994. First appeared as Research Report 119, Digital Equipment Corporation, Systems Research Center.

[15] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[16] Leslie Lamport. Composition: A way to make proofs harder. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference (Proceedings of the COMPOS'97 Symposium)*, volume 1536 of *Lecture Notes in Computer Science*, pages 402–423. Springer-Verlag, 1998.

[17] Leslie Lamport. *Specifying Systems.* Addison-Wesley, Boston, 2002. A link to an electronic copy can be found at `http://lamport.org`.

[18] Leslie Lamport, John Matthews, Mark Tuttle, and Yuan Yu. Specifying and verifying systems with TLA$^+$. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, pages 45–48, Saint-Emilion, France, September 2002. INRIA (Institut National de Recherche en Informatique et en Automatique).

[19] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, May 1999.

[20] Leslie Lamport, Madhu Sharma, Mark Tuttle, and Yuan Yu. The wildfire verification challenge problem. At URL `http://research.microsoft.com/users/lamport/tla/wildfire-␣challenge.html` on the World Wide Web. It can also be found by searching the Web for the 24-letter string `wildfirechallengeproblem`.

[21] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–284, May 1976.

[22] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[23] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[24] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, November 1977.

[25] Serdar Tasiran, Yuan Yu, Brannot Batson, and Scott Kreider. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In *In Proceedings of the 3rd IEEE Workshop on Microprocessor Test and Verification, Common Challenges and Solutions.* IEEE Computer Society, 2002.

[26] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, Berlin, Heidelberg, New York, September 1999. Springer-Verlag. 10th IFIP wg 10.5 Advanced Research Working Conference, CHARME '99.