Model Checking TLA$^+$ Specifications

Yuan Yu, Panagiotis Manolios, and Leslie Lamport

25 June 1999

Appeared in *Correct Hardware Design and Verification Methods (CHARME '99)*, Laurence Pierre and Thomas Kropf editors. Lecture Notes in Computer Science, number 1703, Springer-Verlag, (September 1999) 54–66.

Table of Contents

Model Checking TLA$^+$ Specifications

Yuan Yu[1], Panagiotis Manolios[2], and Leslie Lamport[1]

[1] Compaq Systems Research Center
yuanyu@pa.dec.com, lamport@pa.dec.com
[2] Department of Computer Sciences, University of Texas at Austin
pete@cs.utexas.edu

Abstract. TLA$^+$ is a specification language for concurrent and reactive systems that combines the temporal logic TLA with full first-order logic and ZF set theory. TLC is a new model checker for debugging a TLA$^+$ specification by checking invariance properties of a finite-state model of the specification. It accepts a subclass of TLA$^+$ specifications that should include most descriptions of real system designs. It has been used by engineers to find errors in the cache coherence protocol for a new Compaq multiprocessor. We describe TLA$^+$ specifications and their TLC models, how TLC works, and our experience using it.

1 Introduction

Model checkers are usually judged by the size of system they can handle and the class of properties they can check [3, 16, 4]. The system is generally described in either a hardware-description language or a language tailored to the needs of the model checker. The criteria that inspired the model checker TLC are completely different. TLC checks specifications written in TLA$^+$, a rich language with a well-defined semantics that was designed for expressiveness and ease of formal reasoning, not model checking. Two main goals led us to this approach:

- The systems that interest us are too large and complicated to be completely verified by model checking; they may contain errors that can be found only by formal reasoning. We want to apply a model checker to finite-state models of the high-level design, both to catch simple design errors and to help us write a proof. Our experience suggests that using a model checker to debug proof assertions can speed the proof process. The specification language must therefore be well suited to formal reasoning.
- We want to check the actual specification of a system, written by its designers. Getting engineers to specify and check their design while they are developing it should improve the entire design process. It will also eliminate

> the effort of debugging a translation from the design to the model checker's language. Engineers will write these specifications only in a language powerful enough to express the wide range of abstractions that they normally use.

We want to verify designs of concurrent and reactive systems such as communication networks and cache coherence protocols. These designs are typically one or two levels above an RTL implementation. Their specifications are usually not finite-state, often containing arbitrary numbers of processors and unbounded queues of messages. *Ad hoc* techniques for reducing such specifications to finite-state ones for model checking are sensitive to the precise details of the system and are not robust enough for industrial use. With TLC it is easy to choose a finite model of such a specification and to check it exhaustively.

The language TLA$^+$ is based on TLA, the Temporal Logic of Actions. TLA was developed to permit the simplest, most direct formalization of assertional correctness proofs of concurrent systems [11, 12]. More than twenty years of experience has shown that this style of proof, based on the concept of invariance [2, 15], is a practical method of reasoning about concurrent algorithms. Good programmers (an unfortunately rare breed) routinely think in terms of invariants when designing multithreaded programs.

TLA assumes an underlying formalism for "ordinary mathematics". TLA$^+$ embodies TLA in a formal language that includes first-order logic and Zermelo-Fränkel set theory along with support for writing large, modular specifications. It can be used to describe both low-level designs and high-level correctness properties. TLA$^+$ is described in [13]; recent developments in TLA and TLA$^+$ are posted in [9].

Any language that satisfies our needs will be too expressive to allow all specifications to be model checked. TLC can handle a subclass of TLA$^+$ specifications that we believe includes most specifications of actual system designs.[1] This subclass also seems to include many of the high-level specifications that characterize the correctness of a design. However, it might not be able to handle a large enough model of such a specification to detect any but the simplest errors.

In TLA, specifications are formulas. Correctness of a design means that its specification implies the high-level specification of what the system is supposed to do. The key step in proving correctness is finding a suitable invariant—that is, a state predicate true of all reachable states. Experience indicates that verifying invariance is the most effective proof technique for discovering errors. We believe that it is also the most effective way to find errors with a model checker. TLC can also be used to check step-simulation under a refinement mapping [12], the second most important part of a TLA proof. This requires checking that every step of the implementation is an allowed step of the specification, after appropriate state functions are substituted for the specification's internal variables.

[1] A simple explanation of how TLC evaluates expressions and computes successor states makes it clear what specifications TLC can handle; TLC generates an explanatory error message when it encounters something that it can't cope with.

This substitution is expressed in TLA^+ by an INSTANCE construct, which will be supported only in the next version of TLC. With the current version, the substitution must be done by hand to check step-simulation.

TLC does not yet check liveness properties. We hope to add liveness checking in the future, but we do not yet know if it will be practical for real industrial examples.

The design and implementation of TLC were partly motivated by the experience of the first and third authors, Mark Tuttle, and Paul Harter in trying to prove the correctness of a complicated cache coherence protocol. A two man-year effort generated a 1900-line TLA^+ specification of the protocol and about 6,000 lines of proof—mostly, proofs of invariance. We saw that a model checker could be used to check proposed invariants. Our first thought was to translate from TLA^+ to the input language of an existing model checker. A translator from TLA specifications to S/R, the input language of COSPAN [6], already existed [7]. However, it required specifications to be written in a primitive language that was far from TLA^+, so it was no help. We considered using SMV [14] and Murφ [5], but neither of them were up to the task. Among their problems is that their input languages are too primitive, supporting only finite-state programs that use very simple data types. It was tedious, and in some cases seemed impossible, to translate the mathematical formulas in the TLA^+ specification into these languages. For nontechnical reasons, we did not try to use COSPAN, but we expect that S/R would have presented the same problems as the input languages for the other model checkers. Translating from TLA^+ into a hardware-description language seemed like an unpromising approach.

Because the goal of TLC is finding errors in the specification of an actual design rather than completely verifying a specially-written model, we are willing to sacrifice some speed in order to handle a reasonably large class of TLA^+ specifications. TLC therefore simulates the specification rather than compiling it. It is also coded in Java, rather than in language like C that would be more efficient but harder to program. Despite these inefficiencies, TLC is still acceptably fast. Preliminary tests have found that, depending on the example, TLC runs between two and ten times slower than Murφ, which is coded in C and compiles the specification.[2] TLC is also multithreaded and can take advantage of multiprocessors.

While willing to compromise on speed, we do not want to limit the size of specifications that TLC can handle. Model checkers that keep everything in main memory are usually limited by space rather than time. TLC therefore keeps all data on disk, using main memory as a cache. It makes efficient use of disk with sophisticated disk access algorithms. As far as we know, TLC is the first model checker that explicitly manages all disk access. Murφ [20] uses disk, but relies on virtual memory to handle the state queue, a data structure which contains the unexamined states. As noted by Stern [19] and by us in Section 7, this queue can get very large.

[2] The comparison with Murφ is for a single-threaded version of TLC.

This paper is organized as follows. Sections 2 and 3 describe TLA$^+$ specifications and their TLC models. Section 4 sketches how TLC works. Section 5 describes a compact method of representing states, and Section 6 describes how TLC accesses the disk. Section 7 discusses our initial experience using TLC and presents some preliminary performance data. Finally, Section 8 describes TLC's current status and our plans for it.

2 TLA$^+$

Although TLA is expressive enough to permit a wide variety of specification styles, most TLA system specifications have the form $Init \wedge \Box[Next]_v \wedge L$, where *Init* specifies the initial state, *Next* specifies the next-state relation, v is the tuple of all specification variables, and L is a liveness property written as the conjunction of fairness conditions on actions.[3] TLC does not yet handle liveness properties, so *Init* and *Next* are all that concern us.

We do not attempt to describe TLA$^+$ here, but give an idea of what it is like by presenting small parts of an imaginary system specification. The specification includes a set *Proc* of processors and variables *st* and *inq*, where $st[p]$ is the internal state of processor p and $inq[p]$ is a sequence of messages waiting to be received by p. Mathematically, *st* and *inq* are functions with domain *Proc*. Since TLA$^+$ is untyped, this means that the values of *st* and *inq* in any reachable state are functions. The specification also uses a constant N that is an integer parameter and a variable x whose purpose we ignore. The specification consists of a module, which we name *ImSystem*, that begins:

$$\textsc{extends}\ \mathit{Naturals},\ \mathit{Sequences}$$

This statement imports the standard modules *Naturals*, which defines the set of natural numbers and operations like $+$, and *Sequences*, which defines operations on sequences. A large specification might be broken into several modules. Next come two declaration statements:

$$\textsc{constant}\ \mathit{Proc},\ N \qquad \textsc{variable}\ \mathit{st},\ \mathit{inq},\ x$$

The rest of the specification is a series of definitions. After defining the constant expressions *stInit* and *xInitSet*, the module defines *Init* by

$$\begin{aligned} \mathit{Init}\ \triangleq\ & \wedge\ \mathit{st} = \mathit{stInit} \\ & \wedge\ \mathit{inq} = [p \in \mathit{Proc} \mapsto \langle\,\rangle] \\ & \wedge\ x \in \mathit{xInitSet} \end{aligned}$$

This predicate, which specifies the initial state, asserts that *st* equals the value *stInit*; that $inq[p]$ equals the empty sequence $\langle\,\rangle$, for all $p \in Proc$; and that x may equal any element of the set *xInitSet*. The $\wedge$-list denotes the conjunction of

[3] The specification may also use temporal existential quantification $\exists\!\!\exists$ to hide some variables; such hiding is irrelevant here and will be ignored.

the three subformulas; indentation is used to eliminate parentheses when conjunctions and disjunctions are nested. (We show "pretty-printed" specifications. The actual TLA$^+$ input is an ascii approximation—for example, one types `/\` for $\wedge$ and `\in` for $\in$.)

The bulk of the specification is a sequence of definitions of parts of the next-state relation *Next*. One part is the subaction $RcvUrgent(p)$ that represents the receipt by processor p of the first message of type *Urgent* in $inq[p]$. The action removes the message from $inq[p]$, makes some change to $st[p]$, and leaves x unchanged. A sequence of length n is a function whose domain is $1 \mathrel{..} n$, the set of integers 1 through n. A message is represented as a record with a *type* field that is a string. (Mathematically, a record is a function whose domain is a set of strings, and $r.type$ stands for $r[\text{“type”}]$.) The action $RcvUrgent(p)$ is a predicate on state transitions. Unprimed occurrences of a variable represent its value in the starting state and primed occurrences represent its value in the ending state. The definition of $RcvUrgent(p)$ has the form:

$$
\begin{array}{l}
RcvUrgent(p) \;\triangleq \\
\quad \exists\, i \in 1 \mathrel{..} Len(inq[p]) : \\
\qquad \wedge\; inq[p][i].type = \text{“Urgent”} \\
\qquad \wedge\; \forall\, j \in 1 \mathrel{..} (i-1) : inq[p][j].type \neq \text{“Urgent”} \\
\qquad \wedge\; inq' = [inq \text{ EXCEPT } ![p] = [j \in 1 \mathrel{..} (Len(inq[p]) - 1) \mapsto \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{IF } j < i \text{ THEN } @[j] \text{ ELSE } @[j+1]]] \\
\qquad \wedge\; st' = [st \text{ EXCEPT } ![p] = \ldots] \\
\qquad \wedge\; \text{UNCHANGED } x
\end{array}
$$

Module *ImSystem* defines a number of similar subactions. The next-state relation *Next* is defined to be the disjunction of subactions:

$$Next \;\triangleq\; \ldots \vee (\exists\, p \in Proc : RcvUrgent(p)) \vee \ldots$$

How many subactions like $RcvUrgent(p)$ there are, and how large their definitions are, depend on the system. Here are the sizes (not counting comments) of a few actual specifications:

- A specification of sequential consistency [10] for a simple memory with just *read* and *write* operations is about two dozen lines.
- A simplified specification of the Alpha memory model [1] with the operations *read*, *uncached read*, *partial-word write*, *memory barrier*, *write memory barrier*, *load locked*, and *store conditional* is about 400 lines.
- High-level specifications of the cache coherence protocols for two large Alpha-based multiprocessors are about 1800 and 1900 lines.

3 Checking Models of a TLA$^+$ Specification

Traditional model checking works on finite-state specifications—that is, specifications with an *a priori* upper bound on the number of reachable states. The specification in our example module *ImSystem* is not finite-state because:

- The set *Proc* of processors can be arbitrarily large—even infinite.
- The number of states could depend on the unspecified parameter N.
- The sequences $inq[p]$ of messages may become arbitrarily long.

With TLC, one bounds the number of states by choosing a *model*. In our example, a model instantiates *Proc* with a set consisting of a fixed number of processors; it assigns a particular value to the constant N; and it bounds the length of the sequences $inq[p]$.

To use TLC to check our example, we can create a new module *MCImSystem* that EXTENDS the module *ImSystem* containing our specification. Module *MCImSystem* defines the predicate *Constr*, which asserts that each sequence $inq[p]$ has length at most 3:

$$Constr \triangleq \forall p \in Proc : Len(inq[p]) \leq 3$$

(We could declare a constant *MaxLen* to use instead of 3, and assign it a value along with the other constants *Proc* and N.)

The input to TLC is module *MCImSystem* and a configuration file that tells it the names of the initial condition (*Init*), the next-state relation (*Next*), and the constraint (*Constr*). The configuration file also declares values for the constants—for example, it might assert

$$Proc = \{p1,\ p2,\ p3\} \qquad N = 5$$

These declarations, together with the constraint, define the model that TLC tries to check.[4] Finally, the configuration file lists the names of one or more invariants—predicates that should be true of every reachable state.

TLC explores reachable states, looking for one in which (a) an invariant is not satisfied or (b) deadlock occurs—meaning that there is no possible next state. (Deadlock detection can be turned off.) The error report includes a minimal-length trace that leads from an initial state to the bad state.[5] TLC stops when it has examined all states reachable by traces that contain only states satisfying the constraint. (TLC may never terminate if this set of reachable states is not finite. In practice, it is easy to choose the constraint to ensure that the set is finite.)

In addition to the configuration file and module *MCImSystem*, TLC also uses the modules *ImSystem*, *Naturals*, and *Sequences* imported by *MCImSystem*. The TLA$^+$ *Naturals* module defines the natural numbers from scratch—essentially as an arbitrary set with a successor function satisfying Peano's axioms. A practical model checker will not compute 2+2 from such a definition. TLC allows any TLA$^+$ module to be overridden by a Java class (using Java's reflection) that provides efficient implementations of the operators and data structures defined

[4] TLC can also be used to do nonexhaustive checking by generating randomly chosen behaviors, in which case no constraint is needed.

[5] The trace is guaranteed to have minimal length only when TLC uses a single worker thread. We can easily modify TLC to maintain this guarantee when using multiple worker threads should nonminimality of the trace turn out to be a practical problem.

by that module. TLC provides Java classes for standard modules like *Naturals* and *Sequences*; a sophisticated user could write them for her own TLA$^+$ modules.

4 How TLC Works

TLC uses an explicit state representation instead of a symbolic one like a BDD because:

- Explicit state representations seem to work at least as well for the asynchronous systems that interest us [20].
- A symbolic representation would require additional restrictions on the class of TLA$^+$ specifications TLC could handle.
- It is difficult to keep a symbolic representation on disk.

TLC maintains two data structures: a set *seen* of all states known to be reachable, and a FIFO queue *sq* containing elements of *seen* whose successor states have not been examined. (Another implementation using different data structures is described in Section 6.) The elements of *sq* are actual states, while *seen* contains only the fingerprints of its states. TLC's fingerprints are 64-bit, probabilistically unique checksums [17]. For error reporting, an entry in *seen* also has a pointer to a predecessor state in *seen*. (The pointer is null for an initial state.)

TLC begins by generating and checking all possible states satisfying the initial predicate and setting *seen* and *sq* to contain exactly those states. In our example, there is one such state for each element of *xInitSet*.

TLC next rewrites the next-state relation as a disjunction of as many simple subactions as possible. In our example, the subactions include $RcvUrgent(p1)$, $RcvUrgent(p2)$, and $RcvUrgent(p3)$. (Recall that *Proc* equals $\{p1, p2, p3\}$ in the model.)

TLC then launches a set of worker threads, each of which repeatedly does the following. It removes the state s from the front of *sq*. For each subaction A, the worker generates every possible next state t such that the pair of states s, t satisfies A. To do this for action $RcvUrgent(p1)$, it finds, for each i in the set $1 \mathrel{..} Len(inq[p1])$, all possible values of the primed variables that satisfy the subaction's five conjuncts. (For this subaction, there is at most one i for which there exists a next state t, and that t is unique.) If there is no possible next state t for any subaction, a deadlock is reported. For each next state t that it generates, the worker does the following:

- Check if t is in *seen*.
- If it isn't, check if t satisfies the invariant.
- If it does, add t to *seen* (with a pointer to s).
- If t satisfies the constraint, add it to the end of *sq*.

An error is reported if a next state t is found that does not satisfy the invariant, or if s has no next state. In this case, TLC generates a trace ending in t, or in s if there is no next state t. Using the pointers in *seen*, TLC can generate a sequence of fingerprints of states. To generate the actual trace, TLC reruns the algorithm in the obvious goal-directed way.

A TLA$^+$ specification can have any initial predicate and next-state relation expressible in first-order logic and ZF set theory. Obviously, TLC cannot handle all such predicates. It must be able to compute the set of initial states and the set of possible next states from any given state. Space does not permit a description of the precise class of predicates TLC accepts. In practice, TLC seems able to handle specifications that describe actual systems, but not all abstract, high-level specifications. For example:

– It cannot handle either of the two specifications of sequential consistency in [8] because they are not written in TLA's "canonical form" with a single initial condition and next-state action. The specification $SeqDB2$, with its use of temporal universal quantification, lies well outside TLC's domain. The specification $SeqDB1$ is the conjunction of two specifications in canonical form, which is easy to write in canonical form. (The initial condition or next-state action of a conjunction is the conjunction of the initial conditions or next-state actions, respectively.) TLC could then find simple errors by checking simple invariance properties. However, it could not verify arbitrary invariance properties because the specification is not machine-closed, meaning that the liveness property constrains the set of reachable states. It could therefore probably not find any subtle errors.

– It cannot not handle our original high-level specification of the Alpha memory model [1]. That specification uses a variable *Before* whose value is a relation on the set of all requests issued so far; and it defines a complicated predicate $IsGood(Before)$ which essentially asserts that the results of those requests satisfy the Alpha memory requirements. Actions of the specification constrain the new value of *Before* with the conjunct

  $$(*) \quad (Before \subseteq Before') \ \wedge \ IsGood(Before')$$

  TLC cannot compute the possible new values of *Before* from this expression. However, the formula $IsGood(Before')$ contained the conjunct

  $$Before' \in \text{SUBSET } (Req' \times Req')$$

  where *Req* is the set of possible sequences of requests, and SUBSET S is the set of all subsets of S. Moving this conjunct from $IsGood(Before')$ to the beginning of formula $(*)$ allowed TLC to handle the specification. However, TLC finds possible next values of *Before* by first enumerating all the elements of SUBSET $(Req' \times Req')$, and there are an enormous number of them for any but the tiniest models. In a reasonable length of time, TLC can exhaustively explore only a model with two processors, each of which issues at most one request. Running TLC even on this tiny model led to the discovery of one error in the specification.

  The inability to exhaustively check an abstract specification does not inhibit checking that a lower-level specification implements it. In that case, checking step-simulation just requires that each pair of states that satisfies the lower-level next-state relation also satisfies the higher-level one (under the refinement mapping). For the Alpha memory model, TLC can check this using the original next-state action, without having to rewrite the formula (∗).

- TLC has been applied to, and found errors in, one of the two specifications of cache coherence protocols for Alpha-based multiprocessors mentioned above. That specification was written by engineers with only a vague awareness of the model checker. The only TLC-related instruction they received was to use bounded quantification ($\exists\, x \in S \,:\, P$) rather than unbounded quantification ($\exists\, x \,:\, P$). We believe that TLC will be able handle the other cache coherence protocol specification as well, and we intend to make sure that it does.[6] That specification was written before the model checker was even conceived.

5 Representing States

Because TLA$^+$ allows complex data structures, finding a normal form for representing states is nontrivial. The queue *sq* must contain the actual unexamined states, not just their fingerprints, and it can get very large [19]. We therefore felt that compactness of the normal form was important. The compact method of representing states that we devised could be used by other model checkers that allows complex types, and it is described here.

Our representation requires the user to write a *type invariant* containing a conjunct of the form $x \in T$ for every variable x, where T is a type. The types supported by TLC are based on atoms. An atom is an integer, a string, a primitive constant of the model (like $p1$, $p2$, and $p3$ in our example), or any other Java object with an equality method. A type is any set built from finite sets of atoms using most of the usual operators of set theory. For example, if S and T are types, then SUBSET S and $[S \rightarrow T]$, the set of functions from S to T, are types.

We first convert the value of each variable to a single number. We do this by defining, for each type T, a bijection C_T from T to the set of natural numbers less than $Cardinality(T)$. These bijections are defined inductively. We illustrate the definition by constructing C_T when T is the type $[Proc \rightarrow St]$ of the variable st in our example. In the model, the type of $Proc$ is $\{p1, p2, p3\}$, so we define C_{Proc} to be some mapping from $Proc$ to $0 \,..\, 2$. An element f of $[Proc \rightarrow St]$ is represented as a triple. The jth element of this triple represents $f[C_{Proc}^{-1}(j)]$, which is an element of St. Therefore, f is represented by the triple $C_{St}(f[C_{Proc}^{-1}(0)]), C_{St}(f[C_{Proc}^{-1}(1)]), C_{St}(f[C_{Proc}^{-1}(2)])$. The value of $C_T(f)$ is the number represented in base $Cardinality(St)$ by these three digits.

In a similar fashion, if T is the Cartesian product $T_1 \times \ldots \times T_n$, we can define C_T in terms of the C_{T_i}. Since a state is just the Cartesian product of the values of the variables, this defines a representation of any state as a natural number. This representation uses the minimal number of bits. We use hash tables

[6] The specification uses the TLA$^+$ action-composition operator, which is the only built-in nontemporal TLA$^+$ operator that TLC does not yet implement.

and auxiliary data structures to compute the representation efficiently. The full details will appear elsewhere.

The compact representation did not provide as much benefit as we had expected for two reasons:

– Since the queue *sq* is kept on disk, the only benefit of a compact state representation is to reduce disk I/O. TLC succeeds so well in overlapping reading and writing of the queue with processing that reducing the amount of I/O saved little time.
– The method is optimal for representing any type-correct state, but in real specifications, the set of reachable states is likely to be much smaller than the set of type-correct ones. We found that the queue was not much larger when a simpler representation was used.

We therefore decided that the benefits of our compact representation did not outweigh the cost of computing it, and TLC now represents states with a simpler normal form that can be computed faster.

6 Using Disk

We have implemented two different versions of TLC. They both generate reachable states in the same way, but they use disk storage differently.

The first version is the one described in Section 4 that uses a set *seen* of state fingerprints and a queue *sq* of states. The algorithm implementing the *seen* set was designed and coded by Allan Heydon and Marc Najork. It represents *seen* as the union of two disjoint sets of states, one kept as an in-memory hash table and the other as a sorted disk file having an in-memory index. To test if a state is in *seen*, TLC first checks the in-memory table. If the state is not there, TLC uses a binary search of the in-memory index to find the disk block that might contain it, reads that block, and uses binary search to see if the state is in the block. To add a state to *seen*, TLC adds it to the in-memory table. When the table is full, its contents are sorted and merged with the disk file, and the file's in-memory index is updated. Access to the disk file by multiple worker threads is protected by a readers-writers lock protocol.

The queue *sq* is implemented as a disk file whose first and last few thousand entries are kept in memory. This is a FIFO queue that uses one background thread to prefetch entries and another to write entries to the disk. Devoting a fraction of a processor to these background threads generally ensures that a worker thread never waits for disk I/O when accessing *sq*.

The second version of TLC uses three disk files: *old*, *new*, and *next*, with *old* and *next* initially empty and *new* initially a sorted list of initial states. Model checking proceeds in rounds that consist of two steps:

1. TLC appends to the *next* file the successors of the states in *new*. TLC then sorts *next* and removes duplicate states.

2. TLC merges the *old* and *new* files to produce the next round's *old*. Any states in *next* that occur in this new *old* file are removed, and the resulting *next* file becomes the next round's *new*. TLC sets the next round's *next* file to be empty.

This algorithm results in a breadth-first traversal of the state space that reads and writes *old* once per level. As described thus far, it is the same as an algorithm of Roscoe [18], except he uses regions of virtual memory in place of files, letting the operating system handle the actual reading and writing of the disk. We improve the performance of this algorithm by implementing *next* with a disk file plus an in-memory cache, each entry of which is a state together with a *disk* bit. A newly generated state is added to the cache iff it is not already there, and the new entry's *disk* bit is set to 0. When the cache is full (the normal situation), an entry whose *disk* bit is 1 is evicted to make room. When a certain percentage of the entries have *disk* bits equal to 0, those entries are sorted and written to disk and their *disk* bits set to 1.[7] To reduce the amount of disk occupied by multiple copies of the same state in *next*, we incrementally merge the sets of states written to disk. This is done in a way that, in the worst case, keeps a little more than two copies of each state on disk, without increasing the time needed to sort *next*.

7 Experimental Results

TLC executed its first preliminary test in August of 1998 and has been used to debug specifications. The largest of these is the 1800 line TLA$^+$ specification of a cache coherence protocol for a new Compaq multiprocessor. This specification was written by the engineers in charge of testing. We have used TLC to find errors both in the specification and in a 1000-line invariant for use in a formal correctness proof. TLC has found about two dozen errors in the TLA$^+$ specification, two of which reflected errors in the actual RTL implementation and resulted in modifications to the RTL. (The other errors would presumably not have occurred had the TLA$^+$ specification been written by the design team rather than the testing team.) This specification has infinitely many reachable states. Checking it on a model with about 12M reachable states takes 7.5 hours on a two-processor 600MHz work station, and the state queue attains a maximum size of 250M bytes. The model with the most states that TLC has yet checked, which is for the 30-line specification *CCache* of [8], has over 60M reachable states and takes less than a day to check on a 600MHz uniprocessor work station.

8 Status and Future Work

By using a rich language such as TLA$^+$, we can have the engineers designing a system write a single TLA$^+$ specification that serves as a design document,

[7] The first version of TLC can also be improved in a similar fashion by adding a *disk* bit to the in-memory cache of the *seen* set.

as the basis for a mathematical correctness proof, and as input to TLC. By designing TLC to make explicit and disciplined use of disk, we have eliminated the dependency on main memory, which is the limiting factor of most model checkers. TLC is an industrial-strength tool that engineers are using to help design and debug their systems. TLC was released to other engineering groups in June of 1999. We hope to release TLC publicly in the fall of 1999.

In addition to improving performance and providing a better user interface, possible enhancements to TLC include:

- Checking that the output of a lower-level simulator is consistent with a specification.
- Checking liveness properties.
- Using partial-order methods and symmetry to reduce the set of states that must be explored.

We expect that the experience of engineers using TLC will teach us which of these are likely to have a significant payoff for our industrial users.

Acknowledgments

Homayoon Akhiani and Josh Scheid wrote the specification of the cache coherence protocol to which we are applying TLC. Mike Burrows suggested that TLC be disk based and recommended the second method of using the disk. Damien Doligez and Mark Tuttle are early users who provided valuable feedback. Sanjay Ghemawat implemented the high-performance Java runtime that we have been using. Jean-Charles Gregoire wrote the TLA$^+$ parser used by TLC. Mark Hayden is helping us improve the performance of TLC. Allan Heydon advised us on performance pitfalls in Java class libraries.

References

[1] Alpha Architecture Committee. *Alpha Architecture Reference Manual.* Digital Press, Boston, third edition, 1998.
[2] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.
[3] E.M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 of *LNCS*. Springer-Verlag, 1981.
[4] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.
[5] David L. Dill. The Murφ verification system. In *Computer Aided Verification. 8th International Conference*, pages 390–393, 1996.
[6] Z. Har'El and R. P. Kurshan. Software for analytical development of communication protocols. *AT&T Technical Journal*, 69(1):44–59, 1990.
[7] R. P. Kurshan and Leslie Lamport. Verification of a multiplier: 64 bits and beyond. In Costas Courcoubetis, editor, *Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 166–179, Berlin, June 1993. Springer-Verlag. Proceedings of the Fifth International Conference, CAV'93.

[8] Peter Ladkin, Leslie Lamport, Bryan Olivier, and Denis Roegel. Lazy caching in TLA. *Distributed Computing*, 12, 1999. To appear.

[9] Leslie Lamport. TLA—temporal logic of actions. At URL `http://www.research.digital.com/SRC/tla/` on the World Wide Web. It can also be found by searching the Web for the 21-letter string formed by concatenating `uid` and `lamporttlahomepage`.

[10] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[11] Leslie Lamport. Introduction to TLA. Technical Report 1994-001, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, December 1994.

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

[13] Leslie Lamport. Specifying concurrent systems with tla$^+$. In Manfred. Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, pages 183–247, Amsterdam, 1999. IOS Press.

[14] K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

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

[16] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the 5th International Symposium on Programming*, volume 137 of *LNCS*, pages 337–350, 1981.

[17] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[18] A W Roscoe. Model-checking CSP. In *A Classical Mind: Essays in Honour of C A R Hoare*, International Series in Computer Science, chapter 21, pages 353–378. Prentice-Hall International, 1994.

[19] Ulrich Stern. *Algorithmic Techniques in Verification by Explicit State Enumeration*. PhD thesis, Technical University of Munich, 1997.

[20] Ulrich Stern and David L. Dill. Using magnetic disk instead of main memory in the Murφ verifier. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 172–183, Berlin, June 1998. Springer-Verlag. 10th International Conference, CAV'98.