Exercises in Style (Alpha Specifications)

Uri Abraham and Tamar Pinhas
Departments of Mathematics and Computer Science,
Ben-Gurion University, Be'er-Sheva, Israel

March 28, 2003

Abstract

We compare two documents: the Alpha Architecture Reference Manual (from Compaq Computer Corporation) and the **TLA**$^+$ specification of the Alpha memory model (by Lamport, Sharma, Tuttle, and Yu). We compare the styles, the languages employed, and the fundamental assumptions (that is the basic frameworks of the two texts). We conclude that the **TLA**$^+$ specification is not so much a formalization of the Alpha Reference Manual text as it is a translation into a different framework. We believe that a formalization of the Alpha Reference Manual can be useful, and we propose one.

1 Introduction

The Wildfire Verification Challenge was posted on the World Wide Web at URL

http://research.microsoft.com/users/lamport/tla/wildfire-challenge.html

It is the work of four authors: Leslie Lamport, Madhu Sharma, Mark Tuttle, and Yuan Yu (hereafter Lamport et al.). An article [8] by Lamport et al. describes and explains this challenge. It consists of two documents written in **TLA**$^+$ a specification language designed by Lamport [7] for the specification of concurrent processes. The first document (denoted here **TLA**$^+$ Alpha) is a specification of the Alpha memory model which any Alpha multiprocessor must implement, and the second is a formal definition of a cache-coherence protocol which is supposed to implement this Alpha specification, but contains an error on purpose. The challenge is to find this, and additional, bugs in the **TLA**$^+$ Wildfire protocol definition.

We are not concerned here with the bug or with the Wildfire protocol, but with the **TLA**$^+$ Alpha document which is supposed to be a formal counterpart of the Compaq Alpha Reference Manual [3] (hereafter, called "the Manual"). The relationship between the Manual and the **TLA**$^+$ Alpha is viewed by Lamport et al. as that between an informal specification text and a formal, machine-readable **TLA**$^+$ module which mirrors the informal description. (Section 3 of [8]

refers to the Alpha Reference Manual several times as "informal specification".) We will discuss the term "formalization" and inquire for the precise relationship between these documents. We will argue that the Manual is "semi-formal", and it resembles in this respect regular mathematical texts (books and articles) that are studied by mathematicians. The architects of the Alpha provide an explication of the Alpha machine, namely they describe the basic theoretical concepts needed for its understanding, and provide convincing (albeit not completely formal) proofs of certain properties of the architecture. From a formalization of the Manual we expect a clearer description of the Manual's language, a mathematical definition of the models involved, and additional details needed for a totally formal rendering of the specification. The **TLA**$^+$ Alpha is a formal specification indeed, but it is set in a different framework. While the Manual speaks about events (memory accesses) the **TLA**$^+$ Alpha speaks about states and describe their transitions. Thus the **TLA**$^+$ Alpha translates the Manual into a different language rather than investigates the Manual's original language. Hence (we will argue) a formalization is still missing–and needed.

The Manual that we have at hand is [5], which is the fourth edition (2002) rather than the third edition used by Lamport et al. We assume that the differences are not in the parts investigated. The Manual is a heavy document (19 chapters and several appendixes), so Lamport et al. necessarily simplify and abstract away some of the more complicated issues of the Manual in order to represent the main ideas of the architecture. This is a reasonable approach, and we adopt some of their simplifications in this article. Since we are interested in issues of concurrency, we deal only with chapter 5 of the Manual: "System Architecture and Programming Implications". Actually we concentrate on section 5.6 "Read/Write Ordering" which interests us most.

The plan of the paper is as follows. In Section 2 the style of the Manual is discussed and some of its basic concepts are explained. In Section 3 an assembly language is described. Then in Section 4 we report section 5.6.1 of the Manual in its own words. This is followed in Section 5 by our formalization of the specifications of 5.6.1. Then it is shown how these formal properties imply the correctness of a simple mutual exclusion algorithm (in Section 6). The **TLA**$^+$ Alpha is described in Section 7. The conclusions in Section 8 summarize the main arguments of our paper.

2 The Manual's style

In fact only a small part of the Manual is investigated in this paper. Specifically, we study sections 5.6.1 and 5.6.2 and the term "Manual" usually refers to these two sections.

We discern the following languages in the Manual:

1. Plain English. This is most often professional language, not so easy for the inexperienced. Some parts of the Manual are certainly not intended to be formalized, they provide historic background and other complementary

information which is not, properly speaking, part of the specification. For example "Alpha becomes the first 21st century computer architecture" is not part of the architecture description which interests us. When we say here "plain English" we refer to expressions in English which can be formalized, or which give some hints as to the direction in which a formalization will fit best the intentions of the authors of the Manual. For example, even a plain English nonsense such as "time must not go backward" (from 5.6.2.1 of the Manual) is interesting to us, if only because it shows that the architects were preoccupied with the notion of time despite the fact that it never appears in their formal specifications.

Another example of plain English is from 5.6.1, page 5-11.

> "Each processor may generate accesses to shared memory locations. There are six types of accesses."

It may appear far-fetched now, but we understand the verb "generate" as an indication of some motion, or creation, which involves time. Accesses are events, located in time, and the six types are predicates defined on these events.

2. Assembly language or mnemonic used to define the diversity of operations of the processors. An example of the mnemonic language used in the Manual cam be found on page 5-15, where processor Pi is said to execute the following:

   LDQ R1, x

   STQ R1, y

3. Symbolic language. Compound symbols are of prime importance in the Manual. For example

   Pi:R<size>(x,a)

   which is a symbol with several components. However, these compound symbols do not form a language, they are not associated by grammatical rules. They are rather combined with English as in the following item.

4. Mixture of English and symbolic language. This is the most interesting part, for us, of the document. It is the part which we claim to be "semi-formal" and which can be fully formalized. We will investigate it next.

A word about quotations. To exemplify or to make a point, we report from the Manual. Most often these reports are not exact citations but rather paraphrases of the document. Since we disregard certain operations (such as instruction fetch) we must omit parts of the Manual, which necessarily brings some minor changes but do not alter the meaning of the quoted text (for example grammatical changes). We therefore use the verb rephrase or report so as not to be blamed for misquoting, but our reports are meant to be as accurate as possible in the given context.

The Manual (Section 5.6.1 page 5-11) describes six types of memory accesses. This description is an example of a mixture of English and symbolic language which we want to investigate now. We will explain in details these types later on, now we are interested in analyzing the logical status of some symbols found in this description.

Three of the six types of accesses that are mentioned in 5.6.1 (page 5-11) are reported as follows:

1. Data read (including load-locked) by processor i to location x, returning value a, denoted Pi:R<size>(x,a).
2. Data write (including successful store-conditional) by processor i to location x, storing value a, denoted Pi:W<size>(x,a).
3. Memory barrier issued by processor i, denoted Pi:MB.

Here "Pi:R<size>(x,a)" is an example of a compound symbol. In fact it is a meta-symbol, that is, a description of a collection of symbols obtained by supplementing values for size, x, and a.

What is the status of these "types"? What is, for example, the grammatical role of "Pi:R<size>(x,a)" in the Manual specification language? It is clear that these types are in fact predicates defined on the memory accesses. They qualify the accesses and thus serve as predicates in the Symbolic/English language. For example, look at item 5.6.1.5 Definition of Visibility. It begins as follows:

> If u is a write access Pi:W<m>(x,a) and v is an overlapping read access Pj:R<n>(y,b), etc.

If we write Pi:W<m>(x,a) as a single symbol S and write Pj:R<n>(y,b) as T, and if the relation of overlap is written as *overlapping*, then the quotation above is written in first-order language as

> If $S(u)$ and $T(v)$ and $\mathit{overlapping}(u, v)$ etc.

Another indication that these types are used as predicates is seen in the descriptions of the litmus tests. For example, in 5.6.2.1 Litmus Test 1, we see the expression

> [U1] Pi:W<8>(x,2)

It is clear from the following Manual text that U1 is a memory access, so that this expression is an application of the predicate Pi:W<8>(x,2) to the argument U1. If we write this predicate in one symbol as S, then that formula would be written in traditional logic as S(U1). The decision to write the subject of the predicate to its left and in square brackets is legitimate, and even makes sense. We want to clarify the logical status of the different parts of the language used by the Manual, not to correct it. So when we describe the logical formulas that are, in our opinion, the equivalent of the Manual expression, we write $S(U)$ rather than $[U]S$ without any desire to correct the manual's language or to impose our own language.

If we accept that compound symbols such as "Pi:W<8>(x,2)" serve as predicates, then the following question is immediately posed: what is their domain? Namely, what is the nature of the accesses that are predicated by these symbols? We think that the accesses are events (or rather event occurrences). They are "generated by the processes", so they appear and disappear. These accesses are the basic undefined things of the universe of discourse, like points and lines in geometry. Not only unary but binary relations are defined as well over the accesses. For example, in 5.6.1.2 Definition of Before and After, we read:

> The ordering relation BEFORE ($\Leftarrow$) is a partial order on memory accesses.

The logician sees here immediately an abstract (model theoretic) structure with a universe (the memory accesses) and diverse predicates—some of which are unary and some with larger arity. Another remark: we see here that not only a symbol is defined (namely $\Leftarrow$) but an English equivalent BEFORE is also introduced. In formulas, one can use the symbol, and in a less formal discourse the term BEFORE can be used. The writers of the Manual seek a formal specification, but they also want a flexible English language with which the formal expression could be described. It is important in our opinion not only to devise a formal language for mathematical expressions, but also to enhance the informal language with expressions and terms (such as BEFORE) that have unique formal counterparts.

In formal logic there is a distinction between predicates and formulas. Predicates are the simplest building blocks of which formulas are made: if P is an n-ary predicate and $\tau_1, \ldots, \tau_n$ are terms (expressions that have values) then $P(\tau_1, \ldots, \tau_n)$ is an atomic formula. Composite formulas are build from atomic formulas with the aid of logical connectives and quantifiers. Now we ask the following question: is an expression of the form "Pi:R<size>(x,a)" an indecomposable predicate, or should we view it as a compound formula? If the latter is true, then the formula

> [U1] Pi:R<size>(x,a)

for example, can be construed as a conjunction of [U1]Pi with [U1]R etc. Since the Manual never mentions such sub-formulas, we conclude that Pi:R<size>(x,a) is a single symbol and the above formula is atomic.

This use of compound symbols by the Manual is probably quite efficient, yet in logic there is a tradition of identifying the most elementary parts that can be used as predicates. So that a predicated expression such as [U1]Pi:R<2>(x,a) is seen as equivalent to a conjunctive formula saying about U1 that it "belongs" to process Pi, that it is a Read access, of size 2, at location x, and with value a. Thus a formalization of the Manual should introduce elementary predicates and function symbols with which these compound predicates can be expressed.

It is possible to have for every possible value a a predicate $V_a(u)$ which says that access u has value a. Yet, since there are so many locations and possible values, we introduce them as constants, and then have a function that assign

to every memory access its location and another function for its value. So we prefer to have a function *Value*, and to use the formula $Value(u) = a$ to indicate that a is the value of access u. (And likewise an address function is introduced, and $address(u)$ gives the address of access u.)

Even though we concluded that Pi:R<2>(x,a) is seen by the Manual as a single predicate (for every index i, location x, and address a), it is clear that the Symbolic-English language employed by the Manual contains expressions for each of the minute parts of the predicate. For example, we read "if u is a write access etc." which can be rendered symbolically as "if $Write(u)$ etc." where $Write$ is a predicate on accesses used to denote write events. The fact that the Manual abounds with such English expressions (even though never in symbolic formulas) indicates that they should be in any formalization of the Manual.

Here is a second example. In several places we read in the Manual "processor Pi executes" etc. (e.g. page 5-15). Here Pi serves as a name, a constant, not as a predicate. Or, for another example, on page 5-13 we read

> For two accesses u and v issued by processor Pi, etc.

Here Pi can be construed either as a predicate or as a constant. If it is viewed as a constant, then that sentence can be formalized with the aid of a function "processor" which gives to each access c its "owner" $processor(c)$. Then the above sentence would begin with: if $processor(u) = P_i$ and $processor(v) = P_i$ etc. If, however, Pi is construed as a predicate, then this sentence can be formalized as: if $P_i(u)$ and $P_i(v)$ etc.

We summarize this section with the following conclusion which motivates our subsequent usage of system executions. We have seen that in many places the Manual employs a semi-formal English-Symbolic language. In this language unary and binary predicates appear, which have as universe a collection of events (called memory accesses).

3 A machine language

In order to understand some parts of the Manual's specification (for example the definition of Dependence Constraint 5.6.1.7) we think it necessary to refer to some rudimentary assembly language (a symbolic form of machine language). So we formulate such a language which is based on the information gathered from the Manual, but is simplified as much as possible.

The basic data unit is the byte. A byte is a sequence of bits of length 8, namely a function from $\{0, \ldots, 7\}$ (the set of first eight natural numbers) into $\{0, 1\}$. For every natural number k, a k-word is a sequence of length k of bytes, namely a function w defined on $\{0, \ldots, k-1\}$ such that each $w(i)$ is a byte (for any integer $0 \leq i < k$). Specifically, Alpha has k-words for $k = 2, 4, 8$, and they have special names. A word is a two-byte data unit, a longword lw is a quadruple $\langle lw(0), lw(1), lw(2), lw(3)\rangle$ of bytes, and a quadword is a sequence of eight bytes. If q is a quadword, then for any $0 \leq i < 8$, $q(i)$ denotes the i-th byte of q. We can think of a quadword q as a sequence s of bits of length 64. One

possibility is as follows. For every $0 \leq k < 64$, if $k = b \times 8 + i$ (with $0 \leq i < 8$) then $s(k) = q(b)(i)$.

The memory consists of 2^{64} addresses, and each address holds a byte. So any quadword determines an address (being a sequence of 64 bits). (This description of the memory addresses is quite informal, and is brought here as an introduction. A formal specification will be given at a higher degree of abstraction.)

The assembly language can specify an address and also an interval of 2, 4, or 8 consecutive addresses. If a is an address (a quadword), then $(a, 2)$ specifies the interval of two addresses $a, a + 1$. Similarly, $(a, 4)$ denotes the interval $a, a+1, a+2, a+3$, and $(a, 8)$ is the interval of eight addresses from a to $a+7$. For uniformity, we can write $(a, 1)$ instead of the plain address a. Formally, any interval is a sequence, namely a function. If t is the interval denoted by $(a, 8)$, for example, then $t(0) = a$, $t(1) = a + 1$, and so on until $t(7) = a + 7$. Any interval t can also be seen as a set of addresses (its address range), and sometimes we refer to address intervals as *sets* of addresses. So we can say that address d belongs to address interval t instead of the formal $\exists i\ d = t(i)$.

There is a restriction imposed on the addresses that appear in these interval descriptions: they have to be "naturally aligned". We say that an interval of addresses (a, ℓ) is naturally aligned iff a (as a binary number) is a multiple of ℓ. For example, $(a, 8)$ is naturally aligned iff the 3 lower bits of a are zeros. The point is that if $i = (a, \ell)$ and $j = (b, m)$ are (naturally aligned) address intervals, then either they are disjoint (sets of addresses), or else $i \subseteq j$, or $j \subseteq i$.

There are 32 registers named R_0 through R_{31}. Each register is a quadword. The simplest instructions in our language are assignment instructions of the form $R_i := \varphi$, where φ is an arithmetical expression involving only registers and integer constants. For example

$$R_3 := R_5 \times (R_1 + 2)$$

is an assignment instruction. It assigns to register R_3 the product of (the value of) R_5 with the sum of R_1 and 2 (seen as binary numbers of quadword length).

There are three types of "memory" instructions: read, write, and memory-barrier. And there is a single type of "branch" instruction. Read instructions are further divided into Load and Load_Locked. Write instructions are divided into Store and Store_Conditional. These read and write instructions are the data transfer instructions.

As we have said, address intervals are specified as pairs $(\text{adr}, \text{size})$ where adr is an address (namely a binary quadword) and size is in $\{1, 2, 4, 8\}$. This address interval refers to the interval of addresses $\text{adr}, \ldots, \text{adr} + \text{size} - 1$.

In addition to the register assignments, we have the following instructions in our simple language.

1. Read instructions: A Load instruction has the form

   LD R_i $(\text{adr}, \text{size})$

Where R_i is a register, adr an address and size $\in \{1, 2, 4, 8\}$. Its executions copies the bytes in the address interval into register R_i as follows. Byte adr is copied into byte $R_i(8 - \text{size})$ of the register. Byte $\text{adr} + 1$ (if in the interval) is copied into byte $R_i(8-\text{size}+1)$, and so on until byte adr+size−1 is copied into $R_i(7)$. If size < 8, then the remaining $(8 - \text{size})$ bytes of R_i are not changed. The memory itself is never changed in a read operation.

For example, if size $= 1$ then LD R_i (adr, size) copies the byte adr into the last byte $R_i(7)$ of the register. Thus, if the register is 0 before the load and the value of adr is 1, then after the load the value of R_i is 1.

A Load_Locked instruction has the form

LD_ L R_i (adr, size)

The Load_Locked instruction is a read instruction as the Load, but it also "locks" a block of 16 addresses that includes the address interval. In details, let b be such that $\text{adr} = 16b + s$, where $0 \leq s < 16$. Then the interval (adr, size) (which is naturally aligned) is included in the block $16b, 16b + 1, \ldots, 16b + 15$. This block of 16 addresses is called the locking block of address adr.

2. Write instructions: A Store instruction has the form

   ST R_i (adr, size)

   Where R_i is a register $(0 \leq i < 32)$ and (adr, size) denotes an address interval (always naturally aligned). Its execution writes the content of register R_i into the interval of addresses $\text{adr}, \ldots, \text{adr} + \text{size} - 1$, according to the following rule. The register is always a quadword (eight bytes), but the address interval (adr, size) is possibly smaller. The last byte of the register (denoted $R_i(7)$) is copied into the last byte of the address set (namely into address $\text{adr} + \text{size} - 1$), and in general, for any $0 \leq j < \text{size}$, byte $R_i(8 - \text{size} + j)$ is copied into the byte at $\text{adr} + j$. Any other address byte is left unchanged. The register does not change.

   Store_Conditional writes have the form

   ST_C R_i (adr, size)

   A Store_Conditional execution can be successful or not. If it is successful then the values of the bytes of R_i are transferred to the address interval (adr, size) as in the regular Store, and the value of register R_i after the execution is 1. But if it is not successful then no write occurs and the value of register R_i after the execution is 0.

   A successful Store_Conditional releases the lock on its locking block.

   A detailed and formal description of this operation is given in section 5.

3. Branch instructions have the form

BR R_i *line*

Where *line* is a line number (or tag) of the program executed (see below). This condition checks if register R_i is zero: if it is, then control is transferred to the instruction at the indicated line, but if not then control is transferred to the following instruction.

A branch execution in which R_i is found to be zero releases any locking of a previous *LoadLocked* instruction.

4. Memory Barrier instructions have the form

   MB

   The role of these instructions will be explained later on.

A sequential program (for processor P_i) is a sequence $P = \langle i_0, \ldots i_k \rangle$ of instructions, where instruction i_n is attached to line n, such that the Branch instructions refer only to lines in the integer interval $[0, \ldots, k]$. Instruction i_k is a special *end* instruction whose execution stops the program. A program also includes the assumed initial values of the 32 registers of its processor. Our mission is to define in details the executions of a system consisting of several such programs which are executed concurrently and share the same memory space (but each process with its own program and registers). However in this section we shall be concerned with a single process and its instruction execution model. In this model, the instructions are executed exactly in the order prescribed by the program. In actual executions, this ordering can be relaxed and instructions can be executed in a different ordering which satisfies certain requirements, as we shall see. To express these requirements we need a precise definition of this simple instruction execution model for a single processor. For this we need the notion of a state of P_i.

A state of P_i is a function s that assigns to each register R_j of P_i a quadword $s(R_j)$, and assigns to a special variable called PC (for program counter) a line number: $0 \leq s(PC) \leq k$. (PC is unique to P_i, but we write PC rather than PC_i, just as we write R_3 rather than R_3^i. The point is that the identity of the process will always be clear from the context.)

An execution by process P_i of its program $\langle i_0, \ldots i_k \rangle$ is a sequence

$$E = \langle s_n \mid n \in I \rangle$$

of states of process P_i such that the following holds.

1. I is either the set ω of natural numbers or a finite initial segment of this set.

2. For every $0 \leq r < 32$, $s_0(R_r)$ is the initial value of register R_r, and $s_0(PC) = 0$ refers to the first instruction in the program. Thus s_0 is an initial state.

3. Suppose that $s_m(PC) = \ell$. Then the values of s_{m+1} reflect the idea that instruction i_ℓ is executed in state s_m, and s_{m+1} is the resulting state. In details we have the following. In case $s_m(PC) = k$ is the "end" instruction, m is $\max(I)$ and the execution ends. So assume that $\ell < k$.

   (a) If i_ℓ is an assignment instruction of the form $R_j := \varphi$ where φ is some arithmetical expression involving registers of process P_i and constants, then the only changes are:

      i. $s_{m+1}(R_j)$ is equal to the value of φ as computed at state s_m, and

      ii. $s_{m+1}(PC) = \ell + 1$.

   (b) If i_ℓ is a read instruction (Load or Load_Locked) reading some address interval into register R_j then $s_{m+1}(R_j)$ can be any value that is consistent with $s_m(R_j)$ and the size of the address set accessed. By this we mean the following. If the read instruction is OP R_j (adr, size) where OP is LD or LD_ L, then $s_{m+1}(R_j)(i)$ for i such that size $\leq i < 8$ is any value, but

$$s_{m+1}(R_j)(i) = s_m(R_j)(i)$$

   for $0 \leq i <$ size

   Again control is transferred to the next line, $s_{m+1}(PC) = \ell + 1$.

   (c) Store instructions and MB instructions do not alter the values of the registers, and only the value of PC advances to the following line.

   Store_Conditional instructions of the form ST_C R_i (adr, size), however, leave $s_{m+1}(R_i)$ either in value 1 (the store is then said to be successful) or else in value 0 (and the store is said to fail). In any case the value of PC is increased by 1.

   (d) A branch instruction i_ℓ of the form

$$BR\ R_j\ line$$

   does not change the value of the registers. If $s_m(R_j) = 0$, then $s_{m+1}(PC) = line$, but otherwise $s_{m+1}(PC) = \ell + 1$.

Notice that no memory is mentioned in this simple description, and in particular writes have no effect (on the memory) and reads have no cause. Later, when we combine executions by processes $P_1, \ldots, P_N$ with the specification of the operations this point will be clarified. Since reads can obtain arbitrary values, and store-conditional instructions can fail or succeed arbitrarily, it is clear that there are many possible executions of a program. The role of our definition of an execution of a program is not to give a description of the relationship between the reads and writes of the different processes (this will arrive); the role is rather to define the "dependence" relation in the following subsection.

Let $E = \langle s_m \mid m \in I \rangle$ be an execution of a program P. Any pair $p = \langle s_m, s_{m+1} \rangle$ of consecutive states is said to be a *step* in E. We say that p "executes" instruction $i_{s_m(PC)}$ in program P. Remark that it is possible for two

distinct indices $j \neq j'$ that $\langle s_j, s_{j+1}\rangle$ is exactly the same pair as $\langle s_{j'}, s_{j'+1}\rangle$. Since we are interested in the appearances of the steps, that is in their identity as well as in their place in the sequence, we make the following definition. For every execution $E = \langle s_m \mid m \in I\rangle$ of program P as above, we assume a set A of "events" and a one-to-one enumeration $A = \{a_n \mid n \in I^-\}$, where $I^- = \{i \in I \mid i+1 \in I\}$ (that is, I^- contains all the indices of I except the last index if I has a last index). Then we define for every $a \in A$ its step $step(a)$ by the formula

$$step(a_n) = \langle s_n, s_{n+1}\rangle.$$

Now the objects a in A represent step appearances. The sequence $\langle a_n \mid n \in I^-\rangle$ is called the *issue sequence* of E. If $a, b \in A$, then we define

$$a \ \mathit{ProcessorIssueSequence}\ b$$

iff $a = a_m$ and $b = a_n$ for some $m < n$. So *ProcessorIssueSequence* is a linear ordering of A.

3.1 Dependence relation

Let $E = \langle s_m \mid m \in I\rangle$ be an execution of a sequential program $P = \langle i_0, \ldots, i_k\rangle$, and let A be the corresponding set of events, as defined above. So each a has an appearance index n (that is, $a = a_n$) and a represents the step $\langle s_n, s_{n+1}\rangle$ so that it is an execution of instruction $r = i_{s_n(PC)}$. The *Dependence relation* is defined on the events, that is on A. Intuitively, $a_m \ DP\ a_n$ iff the execution of a_n depends essentially on a_m. For example if a_m is a read into some register R and a_n is the following branch execution that checks if R is zero or not (or a_n is a write execution that uses register R). Another example: a_m is a branch execution, and a_n is a_{m+1} or more generally $n \geq m+1$. That is, a_n is the execution of the instruction to which the branch leads or a later instruction execution.

An example in which $a_m \ DP\ a_n$ does not hold is when a_m is a write event.

Formally we need first define for every step t when it "determines" register R, and when it "uses" register R. In fact, for any byte index $0 \leq i < 8$ we define when step t determines the i-th byte of R, and when it uses that byte.

We say that step t *determines* byte i of register R iff t is an execution of instruction r for which one of the following conditions holds.

A1 r is an assignment instruction of the form $R := \varphi$, or

A2 r is a read instruction of the form OP R (adr, size) where $i <$ size, and OP is either LD or LD_ L

We say that step t *uses* byte i of register R iff t is an execution of instruction r and one of the following three conditions holds.

B1 r is a write instruction that uses byte i of register R, namely a write of the form OP R (adr, size) where $i \geq 8 -$ size and OP is either ST or ST_C.

B2 r is a Branch instruction of the form BR R *line*.

B3 r is an assignment of the form $R_i := \varphi$ where expression φ involves register R.

Now we can define for every two events a_m and a_n in A: a_m DP a_n iff

1. $m < n$ and, for some register R and byte index $0 \leq i < 8$, *step*(a_m) determines byte i of R, *step*(a_n) uses that byte of R, and there is no event a_k where $m < k < n$ and such that *step*(a_k) determines byte i of R. Or else

2. *step*(a_m) is an execution of a branch instruction, and $n \geq m + 1$.

4 The Manual's specification

Following Lamport et al. we simplify the Alpha architecture by referring to a limited set of five basic memory operations (load, load_locked, store, store_conditional, and memory barrier). In this section we quote from Section 5.6.1 of the Manual, called "Alpha Shared Memory Model". However, as explained above, this is an adaptation rather than an exact quotation, since we omit references to operations outside those five basic operations (for example we do not refer to instruction fetch operations). We omit references to I/O devices, and to the distinction between physical and virtual memory. We also provide some notes and comments on the Manual's text (so the word "Note" indicates that the following paragraph is ours and not a quotation).

Our report begins here with page 5-11 of the Manual.

Each processor may generate accesses to shared memory locations. There are three types of accesses:

1. Data read (including load-locked) by processor i to location x, returning value a, denoted Pi:R<size>(x,a).

2. Data write (including successful store-conditional) by processor i to location x, storing value a, denoted Pi:W<size>(x,a).

3. Memory barrier issued by processor i, denoted Pi:MB.

The first two types are collectively called D-stream, or read/write accesses, and they are denoted Pi:OP<m>(x,a), where m is the size of the access in bytes, x is the address of the access, and a is the value representable in m bytes; for any k in the range $0..m-1$, byte k of value a (where byte 0 is the low-order byte) is the value written to or read from location $x + k$ by the access.

The size of a read/write access can be 8, 4, 2, or 1 bytes. All read/write accesses are naturally aligned. That is, they have the form Pi:Op<m>(x,a), where the address x is divisible by size m.

The word "access" is also used as a verb; a read/write access Pi:OP<m>(x,a) accesses byte z if $x \leq z < x + m$. Two read/write accesses Op1<m>(x,a) and

2nd access:	Pi:R<n>(y,b)	Pi:W<n>(y,b)	Pi:MB
1st access ↓			
Pi:R<m>(x,a)	⇐ if overlap	⇐ if overlap	⇐
Pi:W<m>(x,a)		⇐ if overlap	⇐
Pi:MB	⇐	⇐	⇐

Figure 1: Table 5-1: Processor Issue Constraints

Op2<n>(y,b) are defined to overlap if there is at least one byte that is accessed by both, that is, if $\max(x, y) < \min(x + m, y + n)$.

Note: the term "byte" in the expression "a read/write access Pi:OP<m>(x,a) accesses byte z if $x \leq z < x + m$" is misleading, because z is not a byte. It is an address, namely a quadword.

5.6.1.1 Architectural Definition of Processor Issue Sequence

The issue sequence for a processor is architecturally defined with respect to a hypothetical simple implementation that contains one processor and a single shared memory, with no caches or buffers.

Note: We defined in great details (in the previous section) the collection of all possible issue sequences. The Manual is rather terse here.

5.6.1.2 Definition of Before and After

The ordering relation BEFORE (⇐) is a partial order on memory accesses. It is further defined in sections **5.6.1.3** through **5.6.1.9**.

5.6.1.3 Definition of Processor Issue Constraints

Processor issue constraints are imposed on the processor issue sequence defined in Section 5.6.1.1, as shown in Table 5-1:

For two accesses u and v by processor Pi, if u precedes v by processor issue constraint, then u precedes v in BEFORE order. u and v on Pi are ordered by processor issue constraint if the entry in Table 5-1 indicated by the access type of u (1st) and v (2nd) indicates the accesses are ordered.

In Table 5-1, $1st$ and $2nd$ refer to the ordering of accesses in the processor issue sequence.

Table 5-1 permits a read access Pi:R<n>(y,b) to be ordered BEFORE an overlapping write access Pi:W<m>(x,a) that precedes the read access in processor issue order. This asymmetry for reads allows reads to be satisfied by using data from an earlier write in processor issue sequence by the same processor (for example, by hitting in a write buffer) before the write completes. The write access remains "visible" to the read access; "visibility" is described in Sections 5.6.1.5 and 5.6.1.6 and illustrated in Litmus Test 11 in Section 5.6.2.11.

Implementations are free to perform memory accesses from a single processor in any sequence that is consistent with processor issue constraints.

5.6.1.4 Definition of Location Access Constraints

Location access constraints are imposed on overlapping read/write accesses.

If u and v are overlapping read/write accesses, at least one of which is a write, then u and v must be comparable in the BEFORE ordering. That is, either $u \Leftarrow v$ or $v \Leftarrow u$.

All writes accessing any given byte are totally ordered, and any read accessing a given byte is ordered with respect to all writes accessing that byte.

5.6.1.5 Definition of Visibility

If u is a write access Pi:W<m>(x,a) and v is an overlapping read access Pj:R<n>(y,b), u is visible to v only if:[1]

1. $u \Leftarrow v$, or
2. u precedes v in processor issue sequence (possible only if Pi = Pj).

5.6.1.6 Definition of Storage

The value read from any byte by a read access v is the value written by the latest (in BEFORE order) write u to that byte that is visible to v. More formally:

> If u is Pi:W<m>(x,a), and v is Pj:R<n>(y,b), and z is a byte accessed by both u and v, and u is visible to v; and there is no write that is AFTER u, is visible to v, and accesses byte z; then the value of byte z read by v is exactly the value written by u. In this situation u is a source of v.

Notes: The first note is on terminology. We prefer to say that "u is a source of v *for* z" to described this situation. It is possible that a single read has several sources for different bytes that it accesses.

The second note concerns a finiteness condition that has to be introduced. Two versions are given here for the storage property, and the second is supposed to be more formal. Yet there is a slight problem here. From the first version we learn from the definite article in "the latest" that there exists such a write u (and it seems appropriate to assume initial writes of some initial value to each address). Yet in the formal version that follows no such existential commitment is implied. So, for example, if there is an infinite sequence of write events to some address and a read of that address that follows them all, then any value returned by this read will satisfy this formal requirement. This is certainly not what the designers had in mind, and it seems reasonable to make the following assumption about the Processor Issue Sequence ordering and the BEFORE ordering. That any access has only a finite number of predecessors (in each of these orderings). With this assumption it is clear that the two versions of 5.6.1.6 are equivalent.

5.6.1.7 Definition of Dependence Constraint

The depends relation (DP) is defined as follows. Given u and v issued by processor Pi, where u is a read and v is a write, u precedes v in DP order (written $u\ DP\ v$, that is, v depends on u) in either of the following situations:

[1] This should certainly be "if and only if", since A only if B means A implies B.

1. u determines the execution of v, the location accessed by v, or the value written by v.

2. u determines the execution or address or value of another memory access z that precedes v or might precede v (that is, would precede v in some execution path depending on the value read by u) by processor issue constraint.

Note: We didn't understand the meaning of "...precedes v or might precede v (that is, would precede...)".

The dependence constraint requires that the union of the DP relation and the "is a source of" relation be acyclic. That constraint eliminates the possibility of "causal loops".

Note: consider the following example. Suppose three processes X Y and Z. Process X contains a Read event R_a^X and a Write event W_a^X both of address a such that $R_a^X \Leftarrow W_a^X$ because of Processor Issue Constraints. Process Y contains a Read R_a^Y of address a and a Write W_b^Y of address b such that $R_a^Y \, DP \, W_b^Y$. Assume that W_a^X is the source of R_a^Y. And process Z contains a Read R_b^Z of address b and a Write W_a^Z of address a such that $R_b^Z \, DP \, W_a^Z$. Assume that W_b^Y is a source of R_b^Z. Now suppose that W_a^Z is a source of R_a^X. This is clearly counter-intuitive, and it seems to be a causal loop not excluded by the Manual specification.

We conclude from this and from similar examples of causal loops not excluded by the Manual that the dependence constraints requirement is not self-evident, and that the Manual ought to justify it.

5.6.1.8 Definition of Load-Locked and Store-Conditional

For each successful store-conditional v, there exists a load-locked u such that the following are true:

1. u precedes v in the processor issue sequence.

2. There is no load-locked or store-conditional between u and v in the processor issue sequence.

3. If u and v access within the same naturally aligned 16-byte physical and virtual block in memory, then for every write w by a different processor that accesses within u's lock range (where w is either a store or a successful store conditional), it must be true that $w \Leftarrow u$ or $v \Leftarrow w$.

   5.6.1.9 Timeliness

   No write by a processor may be delayed indefinitely in the BEFORE ordering.

5 Formalization of the Manual

Our intention in this section is to show how to formalize that part of the Manual reported in the previous section. Formalization consists in designing a formal

language (the specification language), and writing a list of properties in that language (the specifications). Then basic model theory defines the collection of all interpretation of that language that satisfy the specifications. "Formal language" refers here to the term as it is used in classical logic (rather than temporal logic). That is, a collection of predicates and function symbols (and constants) together with formation rules (which involve logical connectives and quantifiers). Usually we mean first-order logic, but second order can also be used.

The reader has observed that the specification refers a lot to sequences. To give just a couple of examples: a byte is a sequence of length 8 of bits, and an address interval is a sequence of addresses. There may be several approaches to the specification of finite sequences and we have nothing new to say about this subject. We therefore borrow the well-known language of set theory and the standard development of finite sets and sequences within set theory. This language includes a single binary predicate: the membership symbol $\in$, which suffices for the development of all basic set-theoretical notions such as integers and functions. (Of course, equality $=$ is also used.) In fact, there is no need for the full theory (no need for infinite sets), and the universe of hereditarily finite sets, *HF*, suffices. Every set in *HF* is finite, *HF* includes all natural numbers, and is closed under formations of finite subsets. A finite sequence of length k (an integer), for example, is a function defined on all integers i such that $0 \leq i < k$. We are not going to use here any details of set-theory, and in fact we are not really interested in the formalization of standard notions such as sequences. We are interested in the Alpha specification, and so we use a ready made formalization of the basic notions employed (which can be taken from any standard book, e.g. [9]).

Following Barwise [4], We prefer to use that brand of set theory called "set theory with urelements". Urelements are objects that have no members, and yet are not the empty set. Two urelements are distinct despite the fact that they (trivially) have the same extension (the same members). In our case, urelements represent events (such as memory accesses). Now we can have finite sets of events, sequences of events etc.

If U is any collection of urelements, then $HF_n(U)$ is defined by induction on $n \in \omega$ (where ω is the set of natural numbers). $HF_0(U) = U$, and $HF_{n+1}(U)$ is the union of $HF_n(U)$ with the collection of all finite subsets of $HF_n(U)$. Then we define $HF(U) = \bigcup_{n\in\omega} HF_n(U)$. (We assume that no u in U contains a member from $HF(U)$.) Members of $HF(U) \setminus U$ are called sets. That is, the empty set $\emptyset$ and any non-empty member of $HF(U)$ (namely a such that $\exists x(x \in a)$) are sets, but urelements are not. Following Barwise [4], we find it more convenient to keep the urelements and the sets (the empty set together with all non-empty sets in $HF(U)$) as two separate sorts. This means also that there are two types of variables, one for urelements and the other for sets.

The Alpha signature (which is defined below) is a list of predicates and function symbols which will be employed in the formulas that constitute the formalization of the Manual. The language formed by the Alpha signature is a two-sorted language. It consists of two sorts: the *Event* sort and the sets formed

by taking *Event* as urelements (this sort is denoted *Set*). If $\mathcal{S}$ is a structure that interprets the Alpha signature then it has the form $\mathcal{S} = (Event^{\mathcal{S}}, Set^{\mathcal{S}}; F^{\mathcal{S}})$ where $Event^{\mathcal{S}} \cup Set^{\mathcal{S}}$ is the universe of the structure (consisting of the interpretation in S of sorts *Event* and *Set*), and $F^{\mathcal{S}}$ is a collection of relations and functions on the universe that interprets the symbols of the signature.

Definition 5.1 A standard interpretation of the Alpha signature is an interpretation $\mathcal{S}$ such that $Event^{\mathcal{S}}$ is countable and the universe is $HF(Event^{\mathcal{S}})$.

Sorts in a multi-sorted language are akin to types. They serve as unary predicates but they have additional roles. For example, sort *Event* can serve as a predicate: if x is any variable then $Event(x)$ says that x is an event occurrence (by which we mean either a memory access, an unsuccessful store-conditional, a branch, or a local assignment). The advantage of sorts in a multi-sorted language are (1) the possibility of having special variables for each sort, and (2) the possibility of having functions and predicates that are defined only on some sorts. For example, we can decide that the Alpha language assigns variables u, v, w (possibly with indexes) to events. So a formula of the form $\forall u\, \forall v\, \varphi(u, v)$ is equivalent to

$$\forall u\, \forall v\, (Event(u) \wedge Event(v) \rightarrow \varphi(u, v)).$$

The second advantage of two-sorted signature is exemplified by the function index which assigns to every event u the index number $i = index(u)$ such that u is by P_i. In standard logic, functions and predicates must be defined over all elements of the universe of the structure, but this is sometimes unnatural. For example the function $index()$ is naturally defined on the events, assigning to each event its processor's index. But what should be its value on an arbitrary set? In multi-sorted languages, as we have said, it is possible to have functions and predicates that are defined only on specific sorts or tuples of sorts. Of course there are means to handle partial functions also in single-sort languages. For example, by adding a special symbol $\perp$ and requiring that the value of a partial function is this special value whenever it is "undefined". We will also need this special value $\perp$, for example when defining the function *addressInterval* on the read/write memory accesses. This function is not defined on the memory barrier events (which belong to the memory accesses, but have no address) and so we can require that its value on every memory barrier access is $\perp$.

Before writing down the formalization of the Manual, we want to discuss in general terms our intentions. We have in mind N processes $P_1, \ldots, P_N$ that execute concurrently their sequential programs. We view each P_i as a unary predicate defined on the sort of memory accesses. $P_i(x)$ says that event x is in P_i. Then we have for each process P_i a sequential program $Prog_i$ and we want to describe (to specify) all possible concurrent executions of these programs. Each structure (in the family that we are going to define) describes a particular run (execution), and the collection of all such structures is the manifold of all possible runs of the programs. (This is different from some approaches that describe all possibilities in a single structure.) Therefore for every process index i we have in our structure a representation $\langle s^i_m \mid m \in I_i \rangle$ of an execution by P_i

of its program. (See Section 3 for definition of *execution*.) Here I_i is either ω or a finite initial segment of ω. Formally, s^i is a function defined on I_i and, for $m \in I_i$, state s^i_m is the value of this function at m. That is, s^i is a function in the signature of our structures, and s^i_m is just another way of writing $s^i(m)$.

In addition, we have a sort *Event* of events in our structures to represent all events in a run. Each $e \in$ *Event* is in some P_i (except for the initialization events which are considered to belong to "the system" and not to any particular process.) The Read/Write and *MB* events are collectively called "memory accesses", and the predicate *MA* holds exactly for these events. The events that are not memory accesses are either assignment to registers, branch executions, or store-conditional events that are not successful (and are not considered as memory access by the Manual). So each event e in P_i corresponds to some "step" appearance $\langle s^i_m, s^i_{m+1}\rangle$, and each step appearance s is represented by some unique event e such that $s = step(e)$.

Anyhow, starting with *Event* as urelements, we have a sort *Set* consisting of all sets in *HF*(*Event*). We want to represent in our structures the registers and the program counters of the processes, as well as the different instructions and programs of the processes. It is natural to view the registers and program counters as urelements, but we choose to represent them by natural numbers (which are all in *HF*).

In the following, the enumeration of the elements of the Alpha signature is interspersed with minor properties. Then the main properties of the Alpha formalization are stated in this signature.

1. The universe is divided into two (disjoint) sorts: *Event* and *Set*. There is a binary "membership" relation $\in$ defined on the universe. We assume that *Event* is the collection of all urelements, namely those elements of the universe that have no members but are not $\emptyset$. (The empty-set $\emptyset \in$ *Set* is not an urelement). Sort *Set* consists of all sets (the empty-set and all non-empty sets). So basic mathematical notions are unequivocally defined: finite functions, sequences, natural numbers etc.

2. The following formulas are defined on *Set*, and we can use them as predicates.

   (a) *address* is the set of natural numbers adr such that $0 \leq \text{adr} < 2^{64}$. (So *address* is a formula with a free variable adr, and we use it as a predicate, writing *address*(a) to say that a is an address.)

   (b) Byte is the set of all sequences of bits of length 8. Namely the set of all functions defined on $\{0, \ldots, 7\}$ into $\{0, 1\}$.

   (c) *AddressInterval*(ai) iff ai is a naturally aligned sequence of consecutive addresses of length that is one of the values $1, 2, 4, 8$, or 16. (See Section 3 for further details.) If f is an *AddressInterval*, and *address* a is in the range of f, then we say that a is in f. If an address a is both in f and in g (another *AddressInterval*) then we say that f and g are overlapping intervals. We write f *overlaps* g or *overlapping*(f, g).

(d) The notion of a state of P_i can be formally defined in our language, and the following remarks explain how this can be done. The registers $R_0, \ldots, R_{31}$ of P_i can be represented by the pairs $\langle 0, i\rangle, \ldots, \langle 31, i\rangle$, and the programs counter variable, PC, can be represented by $\langle 32, i\rangle$. (Another possibility is to have these state variables as urelements.) Anyhow, we have a set Var_i of all registers and the program counter owned by P_i, and we can define that a state of P_i is a function s defined on Var_i and with values in the appropriate ranges. Clearly, the formula $state(s, i)$ which says that s is a state of P_i can be defined in our language.

3. We assume for each process index i ($1 \leq i \leq N$) a predicate I_i whose extension is either ω (the class of natural numbers) or a finite initial segment of ω. We have in our signature functions s^i defined on I_i and assigning states $s^i(m)$ of process P_i for $m \in I_i$. We can write now in our language a formula which says that each s^i describes an execution (sequence of states) of the program of P_i (see section 3 for the informal definition). Instead of $s^i(m)$ we can write s^i_m.

4. The following are defined on sort *Event*.

   (a) Predicates $P_1, \ldots, P_N$, where N is some constant (a natural number) indicating the number of processes. These predicates are mutually exclusive. The numbers $1, \ldots, N$ are called "process indexes". We also have a function, *index*, such that for every (non-initial) event e $index(e)$ is a process index, and $index(e) = i$ iff $P_i(e)$ (initial events form an exception—they do not belong to any process, see below). Although P_i is a predicate, we say "e is in P_i" as though P_i is a set. Predicate *MA* is defined over *Event*, and if *MA*(e) holds then we say that e is a memory access (event).

   (b) Predicates *Load* and *LoadLocked* are defined on the events. We also define *Read*(x) iff *Load*$(x) \vee$ *LoadLocked*(x).

   (c) Predicates *Store* and *StoreConditional* are defined on the events. A predicate *Successful* is defined on the *StoreConditional* events. (Formally, these predicates are defined on all events. So when we say that a predicate is defined on the *StoreConditional* events we mean that it never holds on events that are not *StoreConditional* events.)
   We also define $Write(x)$ iff

   $$Store(x) \vee (StoreConditional(x) \wedge Successful(x)).$$

   (d) Predicate *MB* is defined on the events. We say that e is a memory barrier access if *MB*(e).

   (e) We say that e is a Read/Write event iff $Read(e) \vee Write(e)$. An event e falls under predicate *MA* iff it is either a Read/Write event, or *MB*(e). Predicates *Store*, *StoreConditional*, *Load*, *LoadLocked*, and

MB are pairwise mutually exclusive. A function *addressInterval* is defined on the Read/Write (and the unsuccessful *StoreConditional*) events. *addressInterval*(e) is in *AddressInterval* and its size is 1, 2, 4, or 8, for every Read/Write e. If e_1 and e_2 are events with intervals $f_k =$ *addressInterval*(e_k) for $k = 1, 2$, then we say that e_1 and e_2 overlap in case the intervals f_1 and f_2 overlap. (We write e_1 *overlaps* e_2 or *overlapping*(e_1, e_2).) If *address*(a) and a is in *addressInterval*(e), then we say that "e accesses a".

(f) Function *Value* is defined on the Read/Write events. If *Read*(r) then *Value*(r) is said to be "the value returned by r", and if $Write(w)$ then *Value*(w) is "the value written by w". If x is a Read/Write event and $\langle a, \ell\rangle =$ *addressInterval*(x), then *Value*(x) is a sequence of ℓ bytes. In plain words, the value, *Value*(x), of any Read/Write event x is a sequence containing the same number of bytes (1, 2, 4, or 8) as the size of the address interval of x. Suppose that x is a Read/Write event and $\langle a, \ell\rangle =$ *addressInterval*(x). If address s is in $\langle a, \ell\rangle$ (namely $s = a + i$ for some $0 \le i < \ell$) then we can write *Value*(x, s) for *Value*$(x)(i)$. Intuitively then, *Value*(x, s) is the value read or written by x on address s.

(g) We assume for every address a an initial *Store* event on that address. These initial events are "system events" they do not belong to any P_i. So the index and *step* functions are not defined on the initial events.

(h) Binary relations $\Leftarrow$, *ProcessorIssueSequence*, and *DP* are defined on *Event*. If $a \Leftarrow b$ then we say that a is BEFORE b.

5. We also have in our signature a function a^i (for each process index i) so that $\{a^i(m) \mid m \in I_i^-\}$ is a one-to-one enumeration of all events that fall under predicate P_i, where $I_i^- = I_i$ if I_i is infinite and I_i^- is $I_i \setminus \{\max I_i\}$ otherwise. $P = \bigcup_{1 \le i \le N} P_i$ is the collection of processors' events, and *Init* = *Event* $\setminus P$ are the initialization events (which are *Store* events). There is a function, *step*, defined on P and we intend to have *step*$(a^i(m)) = \langle s_m^i, s_{m+1}^i\rangle$.

Now we go over Section 5.6.1 of the Manual and write the specifications in the language defined above.

Page 5-11 of the Manual.

The Manual begins with specifying three types of accesses. We render this specification by the following sentence.

$$\forall u\, [MA(u) \Longleftrightarrow Read(u) \vee Write(u) \vee MB(u)].$$

We assume here that variable u is reserved to sort *Event*, so that this sentence says that an event u is a memory access (namely $MA(u)$) if an only if it is a Read, a Write, or a memory barrier. If we substitute for $Read(u)$ and $Write(u)$

their equivalents from items 5(b) and 5(c) above, and we write the resulting formula in English, we get the following: For any event e, e is a memory access if and only if it satisfies one of the following three possibilities.

1. u is a read event, namely a load or a load-locked event.
2. u is a store event, or a successful store-conditional event.
3. u is a memory barrier event.

By comparing this formulation with the Manual's, we can point to some drawbacks of the Manual's wording.

1. The Manual's definition is overloaded. It speaks about locations and values, and it introduces notations such as Pi:R<size>(x,a) which are not needed for this particular definition.
2. Consider, for example, the second item in the Manual's definition which says that the second type of a memory access is:

   > Data write (including successful store-conditional) by processor i to location x, storing value a, denoted Pi:W<size>(x,a).

   Strictly speaking, by including successful store-conditional events, the definition does not exclude unsuccessful ones! It is only by employing intuition that we have interpreted the intentions of the Manual to exclude unsuccessful store-conditional events from the memory accesses.

We believe that experience with formalization can help improve the writing of plain English (technical) texts. It is conceivable that if the writers of the Manual had before them a formal specification, their plain English expression would improve, even if that formal specification remained hidden.

5.6.1.1 Processor Issue Sequence

To define the *ProcessorIssueSequence* and formalize **5.6.1.1** we need some preliminaries. We assume for every process index i that $E = \langle s^i_m \mid m \in I_i \rangle$ is an execution of program $Prog_i$ of P_i (see Section 3). We also assume that the issue sequence $\langle a^i(m) \mid m \in I^-_i \rangle$ is a one-to-one enumeration of all events that are in P_i and that $step(a^i(m)) = \langle s^i_m, s^i_{m+1} \rangle$. So that a^i is a one-to-one enumeration of the events of P_i in their issue ordering.

We now define for any events a, b,

$$a \; ProcessorIssueSequence \; b$$

if and only if

1. $index(a) = index(b)$ (namely, accesses by different processes are incomparable). Say $i = index(a) = index(b)$. Let $a = a^i(m)$ and $b = a^i(n)$. And then we require that
2. $m < n$.

Thus, for every process index i, the restriction of *ProcessorIssueSequence* to the accesses in P_i is a linear ordering, either of type ω or else a finite type. We remark that the Manual defines the *ProcessorIssueSequence* relation only on the memory access events, but we find it natural to include in this relation all events issued by the process (such as branch events).

5.6.1.2: Definition of Before and After

$\Leftarrow$ is a partial ordering of MA. That is, $\Leftarrow$ is transitive and irreflexive, and $a \Leftarrow b$ implies $MA(a) \wedge MA(b)$. Note: this is **5.6.1.2**, but we add the following finite predecessor requirement (compare with **5.6.1.9**):

> for every v in MA, $\{u \mid u \Leftarrow v\}$ is finite.

5.6.1.3 Definition of Processor Issue Constraints

The relation x *ProcessIssueConstraint* y is defined for $x, y \in MA$ if and only if x is an initialization event, or else

> x *ProcessorIssueSequence* y,

and

> x or y (or both) are MB events, or [*addressInterval*(x) *overlaps* *addressInterval*(y) and it is not the case that $Write(x) \wedge Read(y)$].

The requirement of **5.6.1.3** is that for every events x and y,

> if x *ProcessIssueConstraint* y, then $x \Leftarrow y$.

Note that since $\Leftarrow$ is an ordering relation (irreflexive and transitive) we may require that the relation *ProcessIssueConstraint* is an ordering relation, without changing the content of these specifications. (Just take the transitive closure of the relation.) We assume here that if x is any initial store event and y is any non-initial memory access event, then $s \Leftarrow y$ holds. The Manual does not introduce initial *Store* events.

5.6.1.4 Definition of Location Access Constraints

Location Constraints are defined in **5.6.1.4** as the following requirement. If u and v are (distinct) overlapping read/write accesses, at least one of which is a write, then u and v must be comparable in the BEFORE ordering. That is either $u \Leftarrow v$ or $v \Leftarrow u$. This wording of the Manual is already quite formal: we defined *overlapping*(u, v) above, and we defined Read/Write events, so that **5.6.1.4** can be stated in our language.

5.6.1.5 Definition of Visibility

Define the formula $Visible(u, v)$ (in words: u is Visible to v) iff

$$Write(u) \ \wedge \ Read(v) \ \wedge \ overlapping(u, v) \ \wedge$$
$$[(u \Leftarrow v) \vee (u \ \textit{ProcessorIssueSequence} \ v)].$$

Definition of the Storage property 5.6.1.6.

We need a lemma first.

Lemma 5.2 *Suppose that Read(r) and a is an address such that r accesses a. (Namely, define s = addressInterval(r), and then a = s(i) for some i). Then there is a unique write access u that is maximal in the $\Leftarrow$ ordering for which*

$$u \textit{ accesses } a \textit{ and } [(u \Leftarrow r) \vee u \textit{ ProcessorIssueSequence } r].$$

Proof. The statement that u is a maximal event that satisfies property φ means that $\varphi(u)$ and that $\neg\varphi(u')$ for every u' such that $u \Leftarrow u'$. Since $\Leftarrow$ is not a linear ordering, it is not inconceivable that there are two (or more) incomparable maximal events that satisfy a certain property.

In our case, $\varphi(u)$ is the property that $Write(u)$, u *accesses* a, and $(u \Leftarrow r) \vee u$ *ProcessorIssueSequence* r.

Uniqueness is obvious since any two writes that access address a are $\Leftarrow$-comparable (by the Location Constraint 5.6.1.4). Hence it is impossible to find two maximal such writes.

For every address a there exists an initial write that accesses a. By assumption 5.6.1.3, this initial write is BEFORE any read. So the set X of events u that satisfy $\varphi(u)$ is non-empty. X must be finite (by the finite predecessor property for $\Leftarrow$ there are only finitely many events x for which $x \Leftarrow r$, and by the assumption that *ProcessorIssueSequence* is either finite or of order-type ω there are only finitely many events x that satisfy x *ProcessorIssueSequence* r). Thus X has a maximal member in the BEFORE ordering, as required. $\dashv$

We continue with a definition needed for the Storage Property.

Definition 5.3 For every read event v and address a accessed by v, let $u = source(v, a)$ be that unique write event given by the lemma. We say that u "is a source of" v (for a) in this case. When we say that u is related to v in the relation "is a source of" we mean that for some a accessed by v (and u) $u = source(v, a)$.

The Storage Property is the following statement: For every read event v that accesses address a,

$$Value(v, a) = Value(source(v, a), a).$$

5.6.1.7 Definition of Dependence Constraint

We have already defined in details the Dependence Constraint relation on the events in section 3.1. Although we did not give details, it is clear that the instructions of the machine language of Section 3 can be encoded (as numbers for example) and the definition of *DP* can be carried in our language. The Manual requires that the union of the *DP* relation and the "is a source" relation is acyclic. This can be formally stated in our language since we have all the necessary set-theoretical tools for that.

5.6.1.8 Definition of Load-Locked and Store-Conditional

The Manual is very precise here and we only rewrite this section in our words. We first repeat the definition of locking range (assuming that its width is 16 bytes). For every memory access e, if $addressInterval(e) = \langle a, \ell \rangle$, let k and

m with $0 \leq m < 16$ be such that $a = 16 \times k + m$ and then the locking range of a is defined to be the interval $a_0, a_0 + 1, \ldots, a_0 + 15$ where $a_0 = 16 \times k$.

For every event v such that $StoreConditional(v) \wedge Successful(v)$, there exists an event u such that $LoadLocked(u)$ and the following hold:

1. u *ProcessorIssueSequence* v. (So u and v are by the same processor.)

2. For every event x, if $LoadLocked(x) \vee StoreConditional(x)$, then it is not the case that u *ProcessorIssueSequence* x and x *ProcessorIssueSequence* v.

3. Suppose that the lock range of u is the same as that of v. For every w by a different processor from that of u (and v), if $Write(w)$ and w has the same lock range as u, then $w \Leftarrow u$ or $v \Leftarrow w$.

Definition 5.4 Any standard interpretation of the Alpha signature in which the specifying properties described in this section hold is called a "system execution of the Alpha specification." (Standard interpretations were defined in 5.1.)

An advantage of a first-order formalization of the type presented here for the Alpha specification is the possibility of conducting regular mathematical proofs using the axioms and basic definitions of the specification. This is not the case, for example, for temporal logics which are rather specialized. As an example we discuss the possibility for a write W followed by a read R, both by P_i and of the same address, to reverse the issue order so that R is BEFORE W. We prove that this is possible only if the source of R is also in P_i.

Theorem 5.5 *Suppose that W and R are events such that $Read(R)$, $Write(W)$, $index(W) = index(R) = i$, and W ProcessorIssueSequence R. Suppose also that a is an address both in addressInterval(R) and in addressInterval(W) (i. e. W and R both access address a.) If $index(source(R, a)) \neq i$ then $W \Leftarrow R$. (Informally: if the source of R for a is not in P_i then W is BEFORE R.)*

Proof. Let W and R in P_i and address a be as in the lemma. Define $V = source(R, a)$. Then $index(V) = j \neq i$ by the lemma's assumption. Consider Lemma 5.2 and the definition of $source(R, a)$. V is maximal in the $\Leftarrow$ partial ordering in the set of all accesses u such that $Write(u)$, u *accesses* a, and either $u \Leftarrow R$ or u *ProcessorIssueSequence* R. Now W is in this set since W *ProcessorIssueSequence* R. Observe that W and V overlap, since they both access address a. Hence by the Location Constraints (**5.6.1.4**) $W \Leftarrow V$, $V \Leftarrow W$, or $W = V$. Equality is impossible since V and W belong to distinct processes, and different processes are disjoint. But $V \Leftarrow W$ would contradict the maximality of V, and hence

$$W \Leftarrow V$$

is the only possibility remaining. Now $V \Leftarrow R$ and the transitivity of $\Leftarrow$ imply that $W \Leftarrow R$. (The reason for $V \Leftarrow R$ is that $V = source(R, a)$ and V does not stand in the *ProcessorIssueSequence* relation with R.) $\dashv$

A corollary of this lemma is that if W and R are a Write and a Read by process P_i that both access address a, if W *ProcessorIssueSequence* R but $R \Leftarrow W$, then $W = source(R, a)$ or W *ProcessorIssueSequence* $source(R, a)$. Indeed, the lemma says that $V = source(R, a)$ is in P_i, and since the *ProcessorIssueSequence* relation is a linear ordering we must rule out the possibility that V stands in the *ProcessorIssueSequence* relation to W. But in such a case $V \Leftarrow W$ by **5.6.1.3**, in contradiction to the maximality of V.

We continue with a second example to show how regular mathematical reasoning can be conducted within the framework of our Alpha specification. We say that the "inversion phenomenon" is the situation in which a Write W is followed by an overlapping Read R in the *ProcessorIssueSequence* relation and yet $R \Leftarrow W$. We look for an example of a situation described with terms that do not include the $\Leftarrow$ relation and in which the inversion phenomenon is a necessity.

Theorem 5.6 *Suppose in process X the following events are issued in the processor sequence. V_a^X is a Write on address a, followed in ProcessorIssueSequence ordering by another Write W_a^X on the same address a, then a Read $R = R_{a,b}^X$ is issued which is a read of the interval $\{a, b\}$ (consisting of the two addresses a and $b = a + 1$). Now assume another process Y in which the following events are issued in the processor sequence. A Write V_b^Y on address b is followed by a Write W_b^Y on b, which is followed by a memory barrier MB^Y, and finally a Read R_a^Y of address a in Y. Suppose that $W_a^X = source(R, a)$, $V_b^Y = source(R, b)$, and $V_a^X = source(R_a^Y, a)$. Then necessarily $R \Leftarrow W_a^X$.*

Proof. We analyze this situation. First it is clear that

$$V_a^X \Leftarrow W_a^X.$$

(By **5.6.1.3**, V_a^X stands in the *ProcessIssueConstraint* relation to W_a^X, and hence also in the $\Leftarrow$ relation.) A similar argument shows that $V_b^Y \Leftarrow W_b^Y$. The existence of the memory barrier implies that $W_b^Y \Leftarrow R_a^Y$ (by *ProcessIssueConstraint* again).

Since V_b^Y is a source of R and they are in different processes, $V_b^Y \Leftarrow R$ (by the definition of *source* 5.3; since it is not the case that V_b^Y *ProcessorIssueSequence* R). The maximality of V_b^Y excludes the possibility that $W_b^Y \Leftarrow R$, and hence $R \Leftarrow W_b^Y$ (by the Location Constraints they are comparable). Hence $R \Leftarrow R_a^Y$ by transitivity of $\Leftarrow$.

Now if $W_a^X \Leftarrow R$ holds, then $W_a^X \Leftarrow R_a^Y$, and this would contradict the maximality of V_a^X which is the source of R_a^Y for a. Hence $R \Leftarrow W_a^X$ follows. $\dashv$

6 Mutual Exclusion

In section 5.5.3 of the Manual a mutual exclusion algorithm is described. The code of Figure 2 is an adaptation that somewhat simplifies the Manual's algorithm (at a possible price of reduced efficiency, but we want the simplest algorithm). It is written in the language defined in Section 3. The algorithm is

0: LD_ L R_0 $(semaphore, 1)$;

1: BR R_0 0 ;

2: $R_0 := 0$;

3: ST_C R_0 $(semaphore, 1)$;

4: BR R_0 0 ;

5: *MB* ;

6: code of the critical section ;

7: *MB* ;

8: $R_0 := 1$;

9: ST R_0 $(semaphore, 1)$;

10: *end*.

Figure 2: Critical-section code. The initial value of address *semaphore* is 1. The initial value of register R_0 is 0.

based on Dijkstra's well-known semaphore abstraction. An address, *semaphore*, is known to all processes. Its initial value is 1. When a process is in its critical section the value of *semaphore* is 0, but otherwise it is 1. Each process tries to acquire the semaphore, namely to find that it is 1 and to set it to 0 at once. Then the process executes its critical section, and finally, when done, it reset the semaphore, namely it writes it back as 1. This procedure is repeated by each process whenever it wants to access its critical section, however in our simpler presentation we let each process execute this procedure just once. Of course, an Alpha process cannot read and write a register at once, and thus the load locked and store conditional pair of instructions in the protocol.

When process P_i wants to access its critical-section it executes the code of Figure 2. The first instruction is in line 0; it is a load locked instruction which reads address *semaphore* into register R_0. (The initial value of this register is assumed to be 0.) Recall that the pair $(semaphore, 1)$ represents the interval of length 1 beginning with address *semaphore*. So that only *semaphore* is copied into the last byte of R_0 and the first 7 bytes of R_0 remain zero.

The branch instruction at line 1 checks if R_0 is 0 or not. In case it is, the locking is canceled and the process returns to line 0 for another locked load execution. This is repeated until value 1 is obtained in R_0 and then control is transferred to line 2, which contains an assignment instruction that sets the value of R_0 back to 0. Then the store conditional instruction at line 3 is executed. It can either succeed or fail. It is the value of register R_0 after the execution of this store conditional instruction that tells whether it failed or not. The value

$R_0 = 0$ indicates failure, and the value $R_0 = 1$ indicates success. In case of failure, the branch instruction at line 4 sends the control back to line 0, and the cycle described above is repeated. However, in case the branch instruction at line 4 discovers that the value of R_0 is not 0 (indication of success of the store conditional) control continues with line 5. This is a memory barrier instruction. Line 6 represents the critical section, which can be a rather large code (consisting of several sub-lines). It is assumed however that the execution of the critical section always terminates. Then another memory barrier is executed at line 7. The role of these memory barrier instructions before and after the critical section is to ensure that read/write instructions in the critical section are not executed too early (before the successful store conditional), nor too late (after the resetting of the semaphore). At exit, a regular store instruction restores the value 1 of *semaphore*.

6.1 Correctness proof

We assume that there are N processes $P_1, \ldots, P_N$, and we consider a standard interpretation $\mathcal{S}$ in which each P_i executes the critical section code. Consider an execution $E = \langle s_i \mid i \in I \rangle$ by one of the processes. There are two possibilities for E: successful and unsuccessful.

1. An execution E is defined to be successful iff it contains a successful execution of line 3, that is a successful *StoreConditional*. In this case E is finite, and it contains a single successful *StoreConditional* step (it may contain several unsuccessful *StoreConditional* steps that precede it).

2. An execution E is defined to be unsuccessful iff it is not successful. In this case E is infinite, that is, its index set I is ω, and it consists solely of execution steps of lines 0–4, where the executions of line 3 (the *StoreConditional*) always fail, and the branches at line 4 always return the control to line 0. An unsuccessful execution does not contain steps from the critical section code of line 6.

Suppose that E is a successful execution by one of the processes, and let $A = \{a_n \mid n + 1 \in I\}$ be its corresponding set of events with its one-to-one enumeration. By our argument above, there exists a unique event a_n such that $step(a_n)$ is a successful execution of the *StoreConditional* instruction of line 3. We write $a_n = sc(E)$ to denote this unique event. A simple analysis of the algorithm shows that there exists a last *LoadLocked* execution in A that precedes $sc(E)$ (in the *ProcessorIssueSequence* ordering). We denote this event by $\ell\ell(E)$. Then $\ell\ell(E)$ and $sc(E)$ form a u, v pair as in **5.6.1.8**. Clearly the value of $\ell\ell(E)$ is 1. Since $addressInterval(\ell\ell(E)) = \langle semaphore, 1\rangle$, and as $addressInterval(sc(E)) = \langle semaphore, 1\rangle$, it follows that the address intervals of $\ell\ell(E)$ and $sc(E)$ overlap, and hence $\ell\ell(E)$ stands in the *ProcessIssueConstraint* relation to $sc(E)$ (by 5.6.1.3).

Now E contains also an execution of line 6, and we let $CS(E)$ denote this critical section event in A. We consider this event as a memory access.

Since a memory barrier (at line 5) exists between line 3 and 6, we have that $sc(E)$ *ProcessIssueConstraint* $CS(E)$. As $CS(E)$ is terminating, it is followed (in the *ProcessorIssueSequence* ordering) by a memory barrier event (related to line 7) and a *Store* event of value 1 (related to line 9. We let $st(E)$ denote this *Store* event in A. Then we have $CS(E)$ *ProcessIssueConstraint* $st(E)$ (because of this memory barrier event of line 7).

To sum-up, we have the following conclusion. If E is a successful execution by one of the processes, and A is its set of events, then we have marked the following events in A:

$$\ell\ell(E), sc(E), CS(E), st(E). \tag{1}$$

These events are related in the *ProcessIssueConstraint* ordering and hence in the BEFORE ordering from left to right. The value read from the semaphore variable in $\ell\ell(E)$ is 1, the value stored in $sc(E)$ is 0, and the value stored in $st(E)$ is 1. Observe also that the accesses $\ell\ell(E)$ and $sc(E)$ form a u, v pair of *LoadLocked* and *StoreConditional* events as specified in **5.6.1.8**.

Consider two processes, say P_1 and P_2, with successful executions, E_1 and E_2 respectively. Let CS_1 and CS_2 be the two critical-section events in E_1 and E_2. Then $CS_1 = CS(E_1)$ and $CS_2 = CS(E_2)$. We want to prove the mutual exclusion property, which is that $CS_1 \Leftarrow CS_2 \ \vee \ CS_2 \Leftarrow CS_1$. Consider the successful store conditional events $sc_1 = sc(E_1)$ and $sc_2 = sc(E_2)$. It follows from the location constraint that either

1. $sc_1 \Leftarrow sc_2$, or
2. $sc_2 \Leftarrow sc_1$.

We are going to prove that $CS_1 \Leftarrow CS_2$ in the first case, and $CS_2 \Leftarrow CS_1$ in the second case. This is an immediate consequence of the following lemma.

Lemma 6.1 *For every successful execution E, if E' is another successful execution by a different process, then the following holds in $\mathcal{S}$. If $sc(E') \Leftarrow sc(E)$ then $st(E') \Leftarrow \ell\ell(E)$.*

Proof. The lemma thus says that the intervals $[\ell\ell(E), st(E)]$ are always comparable in the BEFORE ordering: if X_1 and X_2 are any such intervals then either $x_1 \Leftarrow x_2$ for every $x_1 \in X_1$ and $x_2 \in X_2$, or $x_2 \Leftarrow x_1$ for any two such events.

Consider the set of writes $\{sc(E) \mid E \text{ is a successful execution in } \mathcal{S}\}$. These writes of zero on *semaphore* are linearly ordered (by the Location Access Constraints **5.6.1.4**). We prove the lemma by induction on this ordering. That is, we prove the lemma for E under the assumption that the lemma holds for every successful execution E_1 such that $sc(E_1) \Leftarrow sc(E)$. (Notice that the number of successful executions is finite.)

So let E and E' be as in the lemma, and assume that

$$sc(E') \Leftarrow sc(E). \tag{2}$$

Since $\ell\ell(E)$ and $sc(E)$ form a u, v pair as in **5.6.1.8**, and as $sc(E')$ is a write by a different process, item 3 of **5.6.1.8** implies that either $sc(E') \Leftarrow \ell\ell(E)$ or $sc(E) \Leftarrow sc(E')$. But the latter possibility contradicts (2) and hence

$$sc(E') \Leftarrow \ell\ell(E) \tag{3}$$

follows. Define $s = source(\ell\ell(E), semaphore)$. Then

$$s \Leftarrow \ell\ell(E)$$

since s is not in the process of E (the sole write access issued by the processor of E are $sc(E)$ and $st(E)$ which are after $\ell\ell(E)$. Compare s and $sc(E')$ (they are comparable in $\Leftarrow$ by Location Constraints **5.6.1.4**). Equality is impossible since s is a write of 1, and $sc(E')$ is a write of 0. $s \Leftarrow sc(E')$ is also impossible by (3), since it would contradict the maximality of s (in the definition of source). Hence

$$sc(E') \Leftarrow s. \tag{4}$$

If s belongs to the process of E', then (4) implies that $s = st(E')$, which proves the lemma. So assume that s is not in E'. Clearly s is not the initial write on *semaphore* (by (4)) and hence it belongs to a third process P'' (different from the process of E and the one of E'). Let E'' be the execution by P''. E'' must be successful (since s is in E'' and unsuccessful executions contain no writes). Hence

$$s = st(E'').$$

Since $s \Leftarrow \ell\ell(E)$, $s \Leftarrow sc(E)$, and it follows that

$$sc(E'') \Leftarrow s = st(E'') \Leftarrow sc(E).$$

Hence the lemma holds for E'' by our inductive hypothesis.

In case $sc(E') \Leftarrow sc(E'')$, $st(E') \Leftarrow \ell\ell(E'')$ follows from the inductive hypothesis, and our lemma ($st(E') \Leftarrow \ell\ell(E)$) is a consequence of some simple transitivity arguments applied to $\ell\ell(E'') \Leftarrow s \Leftarrow \ell\ell(E)$. So assume that

$$sc(E'') \Leftarrow sc(E'),$$

and we shall obtain a contradiction. In such a case $st(E'') \Leftarrow \ell\ell(E')$ follows by applying our lemma to E'. But then $s = st(E'') \Leftarrow sc(E')$ (since $\ell\ell(E') \Leftarrow sc(E')$) contradicts (4)! $\dashv$

7 TLA$^+$ Alpha

TLA$^+$ was created by Lamport and described in several articles (for example [6]).

TLA$^+$ is many things and it can be used in diverse ways. First, it takes seriously the job of writing mathematical formulas that are several pages long. For example, you can give names to formulas and then use these names in more

complex formulas. A traditional mathematical text begin with "let x and y be so and so" and then the text uses x and y in their previously defined sense. In usual first-order logic there is no way to introduce objects in this way, but **TLA**$^+$ has a LET–IN construct that does it.

Set-theory is the basis of mathematics, and it seems to be the basis of **TLA**$^+$ as well (see Note 1 in [6]). So we concentrate on the language of set-theory which is particularly simple since it has a single symbol, the membership predicate $\in$ (and the equality symbol $=$). One can write any set-theoretical formula in **TLA**$^+$ since it has the membership relation built-in.

You are allowed to declare any signature in **TLA**$^+$, namely to declare the vocabulary of your language, and then to write sentences in this language. Say, σ is a signature and φ is a **TLA**$^+$ formula in that signature. The models of φ are universes of set-theory which interpret σ and in which φ holds. Of course, in practice one doesn't need all of set-theory (no need for uncountable sets for example) but the ability to create sets, sequences, and functions is inherent in **TLA**$^+$ since it has all the apparatus of set-theory.

There is a possibility in **TLA**$^+$ of declaring *flexible variables*. Let FV be a set of variables declared to be flexible.

Fix a structure V that interprets signature σ and that is a model of set-theory. A *state* is a map from FV into V. In traditional logic, a state is called an assignment. If ψ is a formula, then $V \models_F \psi$ says that V satisfies ψ with the assignment F.

A *history* is a sequence of states $H = (S_i \mid i \in \omega)$ (say an infinite sequence). Intuitively, a state is a description of an instant. The values of its variables (namely FV) reflects the system at a certain moment. So a sequence of states is a description of a possible evolution of the system in time, when time is discretized. S_0 is the initial state of the system, S_1 is the following state, etc. A pair of states (S, T) is called a step, and a pair $s = (S_i, S_{i+1})$ of adjacent states in H is called a *step* in H. A step is thought of as an atomic change of the system. When the history is an execution of a program, a step is an execution of a single instruction from that program. In applications of **TLA**$^+$, such as in the **TLA**$^+$ Alpha specifications, a state is a global, total description and a history is a complete description of the execution.

We should note the basic difference between these states and the seemingly similar notion in section 3. There we defined the notion of a state of P_i, a single process, but here states are global and describe all the processes in the system. The local states of section 3 gave the minimal information needed to describe the execution of the process's program, and hence the memory or any communication queue was not part of the state.

A main use of **TLA**$^+$ is the specification of histories. **TLA**$^+$ defines a temporal language, and for every sentence τ in that language it defines the relation $H \models \tau$ which says that τ holds in history H. Then the collection of all histories H for which $H \models \tau$ is the system specified by τ.

We give some details on how this is done. **TLA**$^+$ defines for each variable v in FV a primed version v'. A primed version is distinct from any non primed

version. That is, if $FV^{'}$ is the set of all primed versions of variables in FV, then $FV \cap FV^{'} = \emptyset$. One can write formulas in the language of σ containing both unprimed and primed variables.

If S is a state, then S' denotes the corresponding map on $FV^{'}$. That is $S'(v') = S(v)$ for every $v \in FV$. Let ψ be a formula with unprimed and primed variables, and $s = (S, T)$ a step. We say that step s is a "ψ step" If $V \models_F \psi$ where $F = S \cup T'$. That is, we view the pair of states (S, T) as an assignment F of both unprimed and primed variables, defined as follows. For an unprimed variable v, $F(v) = S(v)$, and for a primed variable v', $F(v') = T(v)$. Then we require that formula ψ holds in V under this join assignment.

Now, if $H = (S_i \mid i \in \omega)$ is a history, then we define $H \models \Box\psi$ iff for every $i \in \omega$ $V \models_{F_i} \psi$ where $F_i = S_i \cup S'_{i+1}$. Here $\Box\psi$ is a "temporal" formula which says (intuitively) that ψ holds in every step of H.

TLA$^+$ can define more complex temporal formulas, but for our purpose this short description suffices. Even without going into details, one can appreciate the differences between temporal logic with its histories and the more traditional first-oder (or second-order) logic and its structures (interpretations). The *states* are bona fide members of the *HF*(*Event*) structures (of Definition 5.1). So are the steps and the events $a^i(m)$ that represent them. They are full fledge members of the universe of discourse: they are predicable (subjects of predicates) and can serve as arguments of functions in the structure's signature. Moreover, states, steps, and events can be quantified, and the same quantifiers $\forall$ and $\exists$ can handle uniformly states, events, sets, numbers, etc. In contrast, in any temporal logic structures, the states of a history are not objects on which functions and predicates can be defined. The temporal operators $\Box$ and $\Diamond$ can be seen, in some sense, as replacement for quantifiers, but the states are certainly not members of the temporal-logic histories in the sense that the members of V are.

Another difference to notice is the following. In the temporal logic approach, a state is a complete information of an instant. Module *InnerAlpha* of the **TLA**$^+$ Alpha of Lamport et al., for example, defines variables that reflect in each state the situation at that moment: the requests that were issued so far by each of the processes, the value of each cache location, information on the status of each request etc. That is, **TLA**$^+$ Alpha employs global states. In contrast, the states defined in Section 5 are local. They give only partial information, namely the values of the registers and program counter. The BEFORE ordering for example is not part of this partial information. (A thorough elaboration of this approach to concurrency that uses no global states can be found in [1] and in [2].)

We are not concerned here with the advantages and disadvantages of the different approaches to describe and specify concurrently acting processes, but with the question of formalizing the Manual. Our claim is that the Manual does not speak about global states, that temporal operators are not even hinted in the Manual, and therefore that temporal logic with its histories can translate the Manual into its language but not formalize it.

8 Conclusion

If Alpha were a movie, the Manual would describe it in terms of actions and relations between the actors, but Lamport et al. would analyze it frame by frame, using some language that is appropriate to describe static frames and the minute changes from one frame to the following. Our purpose is to point out that the approaches of the Manual and that of the **TLA**$^+$ Alpha are different. Not because one is informal and the other is formal, but because they speak about different objects and have different approaches to the big question of how to describe reality. The Manual speaks about events–the **TLA**$^+$ Alpha about states; the Manual has functions from events to events–the **TLA**$^+$ has no functions from states to states. What Lamport et al. do is translate rather than formalize the Manual. As in any translation, the question of fidelity arises. The **TLA**$^+$ is formal, no doubts about this, but is it a faithful reproduction of the Manual's intention?

Whenever one takes a text T_1 and formalizes it as F_1 the question of fidelity arises, but if T_1 and F_1 share the same views as to which informal language one should use to describe the universe, then it is much easier to see mistakes or to be convinced about the intuitive correctness of the formalization.

We argue in this paper that an intermediate stage is missing from the translation of Lamport et al., namely a formalization of the Manual within a conceptual framework that is shared both by the Manual and its formalization. Section 5 is an example of what we mean here by a formalization of a semi-formal text within the same framework. It requires a formal definition of a language and a collection of sentences in that language that express (and clarify) the ideas of the text in such a way that the text and its formalization share the same approach. Bypassing this intermediate stage may lead to errors in translation. If the translation is based on a formal intermediate stage, then it is easier to spot errors, and in fact it is possible to formally prove the correctness of the translation. In our opinion, the architects of the Alpha could themselves be the formalizers of the Manual. It would have been much easier for them than learning the **TLA**$^+$ language and writing the specification in that form. As quick as the engineers are to learn the **TLA**$^+$ language, it might be advised to leave the translation process to experts, and to require from the engineers and architects "only" to formalize the properties of their system in their own language.

In a private communication, Leslie Lamport argued that the **TLA**$^+$ specification had a real purpose connected with the verification of a particular cache-coherence protocol (the Wildfire protocol, a simplified version of which appears in the challenge problem.) So, comparing it with a specification written to formalize, in the most direct fashion possible, the informal specification in the manual is something of an apples versus oranges comparison. Engineers, he remarked, would not prefer the formalization of the manual given here to the manual itself. A formal method is irrelevant to the concerns of engineers unless it provides them with a tool they can use to debug their designs. Engineers are interested in **TLA**$^+$ because writing their designs in it allows them to debug them with the model checker.

References

[1] U. Abraham, *Models for Concurrency,* Gordon and Breach, 1999.

[2] Concurrency without global states: a case study. In preparation.

[3] Alpha Architecture Committee. *Alpha Architecture Reference Manual.* Digital Press, Boston, third edition, 1998.

[4] J. Barwise, *Admissible Sets and Structures.* Springer 1975.

[5] Compaq. *Alpha Architecture Reference Manual.* Compaq Computer Corporation, fourth edition, 2002. (Available at ftp.compaq.com/pub/products/alphaCPUdocs/alpha_arch_ref.pdf.)

[6] L. Lamport. The Temporal Logic of Actions. ACM Transactions on Programming Languages and Systems, Vol. 7, No. 7, pp. 1–52, 1993.

[7] L. Lamport. Specifying concurrent systems with TLA$^+$. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational system Design,* pages 183-247, Amsterdam, 1999. IOS Press.

[8] L. Lamport, M. Sharma, M. Tuttle, and Y. Yu. The Wildfire Challenge Problem. 2001.

[9] Y. N. Moschovakis, *Notes on Set Theory.* Springer 1994.