Assertional and Behavioral approaches to concurrency*

Uri Abraham, Ben-Gurion University, Beer-Sheva, Israel

Abstract

We compare two proofs of the mutual-exclusion property of the well known critical section algorithm of Peterson: an assertional proof and a behavioral one. The accepted view is that behavioral proofs are informal and are, for some intrinsic reason, error prone. We try to present a different view and to outline a framework within which the behavioral approach can be formalized in a way that keeps the intuitive content of the behavioral reasoning.

1 Introduction

Behavioral reasoning is useful because it gives insight, but it is prone to errors.

A. U. Shankar, An Introduction to Assertional Reasoning, 1993.

Whereas the invariant–assertional method is well defined, and its formal development is a well established field of pure and applied information science, the term *behavioral reasoning* (or *operational reasoning*) is only loosely used to denote a certain pre-formal activity which belongs, some would say, more to psychology than to computer science. Without denying the proven industrial and theoretical value of the assertional reasoning, I argue that this strong and successful scientific activity has neglected the need to develop the behavioral approach, not as a substitute to the assertional method, but as a way to explain algorithmic ideas and to develop error free distributed algorithms (which may finally be analyzed with assertional methods). My point is that, since there is ample evidence that behavioral reasoning is extensively employed (for example in classes, by the researchers who try to understand or develop a distributed algorithm, and even in publications), it is not enough to send these users to study Latin, but we must also

*After a lecture given at Taras Shevchenko National University, Kyiv, in July 2011

P repeats forever	Q repeats forever
1_P $Remainder_P$;	1_Q $Remainder_Q$;
2_P interested$_P$:= *true*;	2_Q interested$_Q$:= *true*;
3_P turn := Q;	3_Q turn := P;
4_P `await:` ¬interested$_Q$ ∨ turn = P;	4_Q `await :` ¬interested$_P$ ∨ turn = Q;
5_P critical-section;	5_Q critical-section;
6_P interested$_P$:= *false*;	6_Q interested$_Q$:= *false*;

Figure 1: Peterson's mutual exclusion algorithm with an `await` statement. The initial value of interested$_P$ and interested$_Q$ is *false*.

show them ways to adroitly use their own language. We propose in this paper some steps in this direction.

Following Shankar in his paper cited above, we take the well-known critical section algorithm of Peterson [5] and prove in both methods, the assertional and the behavioral, that it satisfies the mutual-exclusion property. There is no need to dwell too long on the assertional approach, since the excellent introduction of Shankar [6] is still recommended and since this is the approach that is universally taken in textbooks and lecture notes, but something has to be said on which we can base our comparison of the methods, and so the following section is a brief description of the assertional method as it applies to Peterson's algorithm.

2 Peterson's algorithm in the assertional approach

Peterson's two process critical section algorithm [5] is presented in Figure 1. The two processes are P and Q and they communicate by means of three serial registers. interested$_P$ and interested$_Q$ are single writer boolean registers that are written by P and Q respectively and can be read by both processes, and turn is a common register on which the two processes may write, and its value are the two tokens P and Q. The initial value of interested$_P$ and interested$_Q$ is false, and the initial value of turn is of no importance.

Peterson's mutual exclusion algorithm is composed of two sequential protocols, one for process P and the other for process Q (which is obtained in a sym-

metric way by changing the names of P and Q). Each process executes its protocol whenever it wants to access its critical section, and so we can think of the activity of P as an alternation of $Remainder_P$ activity (which may be non-terminating) and protocol execution, which consists of execution of lines 1_P up-to 6_P. The protocol that P has to execute in order to enter its critical section is quite short. It first writes the value true in its register $\mathsf{interested}_P$, then it writes the value Q in register turn, and then it waits for the system to announce that condition $\neg\mathsf{interested}_Q \vee \mathsf{turn} = \mathsf{P}$ holds. If and when this condition holds, process P can enter its critical section (line 5_P). For the liveness property we have to assume that it stays in its critical section only a limited amount of time, and on exit from the critical section P resets its register $\mathsf{interested}_P$ to false and returns to its *Remainder* activities.

The assertional method depends on the notion of *global states*. We emphasize this point since the behavioral method needs only local states of the participating processes as we shall see. Generally speaking, a global state is a function from the set of state variables which gives to every variable a value in its type.

- The state variables of Peterson's algorithm are: PC_P, PC_Q, $\mathsf{interested}_P$, $\mathsf{interested}_Q$, turn. The type of PC_P (its set of possible values) is the set of line numbers of the protocol of P: $\{1_P, \ldots, 6_P\}$, and the type of PC_Q is the set $\{1_Q, \ldots, 6_Q\}$. (PC_P and PC_Q are the program-counters of P and Q respectively.) The type of $\mathsf{interested}_P$ and $\mathsf{interested}_Q$ is the set {true, false} of boolean values, and the type of turn is the set {P, Q}.

- A *state* (of Peterson's algorithm) is a function S defined over the state variables which respects the type of each variable.

- An initial state is a state S so that $S(PC_P) = 1_P$, $S(PC_Q) = 1_Q$, and $S(\mathsf{interested}_P) = S(\mathsf{interested}_Q) = \mathsf{false}$.

A state is also a structure. A structure S has a *satisfaction* relation: $S \models \alpha$ says that statement α holds in S.

For example, $\neg(PC_P = 5_P \wedge PC_Q = 5_Q)$ is the "Mutual Exclusion" statement which is denoted ME. It says intuitively that it is not the case that both P and Q are in their critical sections at the same time. $S \models \neg(PC_P = 5_P \wedge PC_Q = 5_Q)$ just means that $S(PC_P) \neq 5_P \vee S(PC_Q) \neq 5_Q$.

States are descriptions of the system that are frozen in time. Change is described by steps. A *step* (also called a *transition*) is a pair of states (S, T) that represents an atomic action (a move or event) by which state S changes to state T. In our example, the step represent executions of the program instructions by process P or Q. For lines numbers i and j (in the protocol of P or of Q), an (i, j)

step is a pair of states (S, T) which represents an execution of the instruction at line i which moves the control from line i of the code to line j.

For example a $(4_P, 5_P)$ step is a pair of states (S, T) so that

- $S \models PC_P = 4_P \wedge (\neg \mathsf{interested}_Q \vee \mathsf{turn} = P)$
- $T \models PC_P = 5_P$,
- for every state variable v other than PC_P, $S(v) = T(v)$.

The last definition that the assertional method requires is that of execution or history. A *history* is a sequence H of states $S_0, S_1, \cdots$ (which we take here to be infinite) so that

- S_0 is an initial state,
- for every index i, (S_i, S_{i+1}) is a step.

A history is a description of a possible execution of the algorithm. In a history H the steps of the two processes *interleave*. To prove the mutual-exclusion property is to prove that in any history H of Peterson's algorithm, each state S_i in the history satisfy the mutual-exclusion sentence ME.

In general, the correctness of a distributed algorithm is expressed by some property φ of sequences of states. (Some sequences may have this property and some perhaps have not.) Assuming that φ reflects what we think is a good behavior of the system, to prove that a distributed algorithm is correct is to prove that all of its histories have property φ. Mathematical formality requires that properties are expressed in some well-formed language and that the satisfaction relation "H satisfies φ" (where H is a history execution and φ a property) is well defined. For the assertional method, this well-formed language is a temporal logic language, but for our purpose (of describing two methods) there is no need to enter into any of the details of these languages and we refer the reader to [6] for example. We note however that the temporal logic languages do not quantify over the events (step occurrences) that the history generates.

The language that the states defined in our example support is a simple propositional language, but states can be more complex structures which interpret quantification languages. These quantifiers, however, only quantify over the members of the state, and they do not quantify over the steps. As we shall see, this is a marked difference between the assertional method and the behavioral, and a main reason for the attractiveness of the behavioral approach is that quantification over the events is possible.

We finally describe the invariant-inductive method. Given a history $H = S_0, S_1, \ldots$ we want to prove that some state statement α holds for every S_i. Like

a proof by induction on i, it suffices to prove that $S_0 \models \alpha$, and that for every i, if $S_i \models \alpha$ then $S_{i+1} \models \alpha$.

The assertional methods consists in proving that α is an `invariant`, which means that:

- If S is any initial state then $S \models \alpha$.
- If (S, T) is any transition step and if $S \models \alpha$ then $T \models \alpha$ as well.

Let's exemplify this by considering the following invariant assertion of Peterson's Algorithm $A_P \wedge A_Q \wedge B_P \wedge B_Q \wedge C$ where:

A_P : $\mathsf{interested}_P \equiv PC_P = 3_P, 4_P, 5_P, 6_P$.

A_Q : $\mathsf{interested}_Q \equiv PC_Q = 3_Q, 4_Q, 5_Q, 6_Q$.

B_P : $PC_P = 5_P \rightarrow \neg\mathsf{interested}_Q \vee PC_Q = 3_Q \vee \mathsf{turn} = P$.

B_Q : $PC_Q = 5_Q \rightarrow \neg\mathsf{interested}_P \vee PC_P = 3_P \vee \mathsf{turn} = Q$.

C : The constants are all different (the line numbers and the token "P" and "Q").

We used here an abbreviation: $X = v_1, \ldots, v_n$ instead of $X = v_1 \vee X = v_2 \vee \cdots X = v_n$.

What are we supposed to do with the assertions A_P, A_Q, B_P, B_Q, C?

- To prove that their conjunction imply the mutual-exclusion statement ME.
- To prove that they hold in every initial state.
- To consider every (i, j) step (S, T) in turn and to prove that if S satisfies all of these assertions, then T satisfies them all as well.

The algorithm of Peterson that we consider and its invariants are so simple that we can leave this mission to the interested reader and continue with a description of the behavioral approach.

3 The Behavioral approach, an introduction

> *Operational reasoning has value. Again, being very subjective, we have found that the flash of insight that sparks the creation of an algorithm is often based on operational, and even anthropomorphic, reasoning. Operational reasoning by itself, however, has gotten us into trouble often enough that we are afraid of relying on it exclusively. Therefore we reason formally about properties of a program, using predicates about* all *states that may occur during execution of the program.*
>
> K. M. Chandy and J. Misra [4] (page **xi**) (Operational in this quotation is what we call Behavioral).

As we saw, the main notions that the assertional method studies are the states, steps, and histories. The behavioral approach, on the other hand, concentrates on the *events*, their temporal precedence relation, and on other predicates and functions over the events. Quantification over the events can be freely used in the behavioral approach. We shall investigate here again the critical section algorithm of Peterson, but now we prefer to replace the `await` statement with an explicit "busy reading" while loop. There is no deep reason for this replacement, but after some thought we decided that this is slightly more suitable for our purpose. The "busy reading" Peterson's algorithm is in Figure 2.

Before line 4_P is executed local variables a and b are assigned values that make $a \wedge (b \neq \mathsf{P})$ true. So the loop 4_P (its body) is executed at least once. Each execution is a read of either register $\mathsf{interested}_Q$ or turn. (For the liveness property it is important that no register is neglected, but for the mutual exclusion property it is not.)

In this section we describe a semi-formal correctness proof for the algorithm of Peterson, and in the following section we describe a possible direction in which formalization may be obtained. Although its main objects of interest are the events, the behavioral method still needs states and transitions in order to convey the semantics of its protocols. But now only local states, local steps, and local histories of each process are needed as we shall see. Before we define the local semantics of the processes and the way in which they can be combined to global system executions, we want to present the behavioral correctness proof in its more intuitive reasoning lines. An advantage of the behavioral reasoning is that, just like any mathematical proof, it can be presented at an informal level and then supplied with more details and be turned into a formal proof *without losing the guidelines of the informal (or semi-formal) reasoning*.

The behavioral approach differs from the assertional one already in the formulation of what is the aim of the proof. Whereas the assertional proof wants to

P repeats forever	Q repeats forever
1_P $Remainder_P$;	1_Q $Remainder_Q$;
2_P interested$_P$:= $true$;	2_Q interested$_Q$:= $true$;
3_P turn := Q; a := true; b := Q;	3_Q turn := P; a := true; b := P;
4_P while: $a \wedge (b \neq$ P) do $4_P(1)$ a := interested$_Q$; or $4_P(2)$ b := turn;	4_Q while: $a \wedge (b \neq$ Q) do $4_Q(1)$ a := interested$_P$; or $4_Q(2)$ b := turn;
5_P critical-section;	5_Q critical-section;
6_P interested$_P$:= $false$;	6_Q interested$_Q$:= $false$;

Figure 2: The critical section algorithm of Peterson with an explicit while loop instead of the `await` statement. Initially interested$_P$ and interested$_Q$ are false.

show that no state violates the mutual exclusion property (in any possible history) the behavioral proof is about the events that an arbitrary execution generates, and it uses the precedence relation in its formulation. The mutual-exclusion property ME is now stated as follows:

If X and Y are critical-section events, then $X < Y$ or $Y < X$.

More formally

$$\forall X, Y\ (CS(X) \wedge CS(Y) \rightarrow X < Y \vee Y < X).$$

We see here how events are quantified and how a predicate, CS, is applied to event variables X and Y.

Our plan for the behavioral proof is as follows. We shall first define some simple "basic properties" that executions of the algorithm possess, and then we will prove that the mutual exclusion property ME follows from these properties. The description of these basic properties and the argument that they hold in every execution is done at a very informal level. But the properties themselves are written (or can easily be written) in a formal first order language that has a clear and definite meaning. The proof that these properties imply the mutual exclusion property is done in a completely formal (or formalizable) way. Finally, in Section

4 we will show how these basic properties can be formally proved to hold in every execution of the algorithm.

Consider an execution of Peterson's algorithm of Figure 2, and let c be some critical-section event in this execution, for example by process P. Then there has to be a successfully terminating execution of the while loop of line 4_P that enabled the entry of P into its critical-section. So there has to be a successful read, which we denote $sr(c)$, so that one of the following two possibilities holds.

1. $sr(c)$ is a read of register $\mathsf{interested}_Q$ with value false, or

2. $sr(c)$ is a read of register turn with value P.

Continuing our inspection of the protocol of process P, we see that, before this successful execution of the while loop, process P had to write in register turn the value P. We denote this write event that corresponds to line 3_P with $w_{turn}(c)$. Then, before $w_{turn}(c)$ we have a write in register $\mathsf{interested}_P$ of the value true (an execution of line 2_P). We let $w_{interested}(c)$ denote this write event. The events that we have defined are ordered (in time) as follows.

$$w_{interested}(c) < w_{turn}(c) < sr(c) < c. \tag{1}$$

Of course, these definitions and properties apply to cs events by process Q as well (changing the role of P and Q).

We shall prove the following theorem rather informally, and when we find that our argument requires additional assumptions we shall write them down [in square brackets] in order to finally gather a complete list of all assumptions that are needed for the theorem to work.

Theorem 3.1. *Let c_1 and c_2 be two cs events by the two processes. If $w_{turn}(c_1) < w_{turn}(c_2)$ then $c_1 < c_2$.*

Before proving this theorem, we note that it implies the mutual exclusion property. Indeed, if c_1 and c_2 are by the same process P or Q then the conclusion follows immediately from the assumption that each of the processes is serial. [Add an assumption on the seriality of every process.] If c_1 and c_2 are by different processes, then we consider their corresponding write events $w_1 = w_{turn}(c_1)$ and $w_2 = w_{turn}(c_2)$. [Add an assumption by which we can conclude that $w_1 \neq w_2$.] So $w_1 < w_2$ or $w_2 < w_1$. [Add an assumption that read and write events in the same register are linearly ordered.] If $w_1 < w_2$ then the theorem implies that $c_1 < c_2$, and otherwise $w_2 < w_1$ implies that $c_2 < c_1$. [Do not forget to write down the properties of the $<$ relation.]

So we now turn to the proof of the theorem. Assume without loss of generality that c_1 is by process P and c_2 by Q. Apply (1) to get that $w_{interested}(c_1) < w_{turn}(c_1)$ and that $w_{turn}(c_2) < sr(c_2) < c_2$. Hence, as we assume that $w_{turn}(c_1) < w_{turn}(c_2)$,

$$w_{interested}(c_1) < w_{turn}(c_1) < w_{turn}(c_2) < sr(c_2) < c_2.$$

Consider the two possibilities for $sr(c_2)$, a read of turn of value Q, and a read of $\mathsf{interested}_P$ of value false, and we shall obtain in both cases that $c_1 < sr(c_2)$, and hence that $c_1 < c_2$ holds.

1. Assume that $sr(c_2)$ is a read of turn of the value Q. Since $w_2 = w_{turn}(c_2)$ is a write in turn of value P, and as $w_2 < sr(c_2)$, the fact that $sr(c_2)$ is a read of value Q [which is different from P] implies that there has to be some write w' in turn of value Q such that $w < w' < sr(c_2)$. [This requires some assumption on the behavior of the read/write events.] But any such write event w' is by process P [because only that process writes this value], and we know that there is no write in register turn by process P between $w_{turn}(c_1)$ and c_1 [we need this property]. So $c_1 < w'$ follows from the seriality of process P, and thus $c_1 < w' < sr(c_2)$ implies $c_1 < sr(c_2)$ as required.

2. Assume next that $sr(c_2)$ is a read of $\mathsf{interested}_P$ of value false. Since $v = \mathsf{interested}_P(c_1)$ is a write on $\mathsf{interested}_P$ of value true, and as $v < w_{turn}(c_1) < w_{turn}(c_2) < sr(c_2)$ implies that $v < sr(c_2)$, the fact that $sr(c_2)$ is a read of value false indicates the necessity of some write v' of value false in register $\mathsf{interested}_P$ so that $v < v' < sr(c_2)$. But there is no write in register $\mathsf{interested}_P$ between $w_{interested}(c_1)$ and c_1, and hence $c_1 < v'$ which implies again that $c_1 < sr(c_2)$.

□

We are now ready to formulate a detailed and complete list of properties from which the theorem that we outlined above follows. We need first to define the language in which these properties are written. The language L_{BP} (BP is for Basic Properties) is a two-sorted language with two sorts: Event and Atemporal. (The events will be the read/write and cs events, and the atemporal members will be the values

$$\{\mathsf{true}, \mathsf{false}, \mathsf{P}, \mathsf{Q}, turn, interested_P, interested_Q, \perp\}$$

(which are are all different). The role of $\perp$ is to be an "undefined" value for partial functions.) We will have the following relations and functions.

1. $<$ is a binary relation on the Event type. P, Q, *read*, *write*, and cs are unary predicates defined on the Event type.

2. *Val* and *register* are functions from the Event type to the atemporal values.

3. sr, w_{turn}, $w_{interested}$ are functions from the Event type into the Event type.

The following examples illustrate how the language L_{BP} can be used. We reserve variables x, y, a, b, c etc. to vary over the events. For example $\forall a((Q(a) \vee P(a)) \wedge \neg(Q(a) \wedge P(a)))$ says that every event is either in Q or in P but not in both processes. The function $register(x)$ gives the register (name) on which x operates. For example, $\forall x(register(x) = \mathsf{turn} \rightarrow Val(x) = \mathsf{P} \vee Val(x) = \mathsf{Q})$ says that the value of any event on register turn is either P or Q. Here is how we say in L_{BP} that for every cs event c by P, $w_{turn}(c)$ is a write on turn of value true. $\forall c(cs(c) \wedge P(c) \rightarrow write(w_{turn}(c)) \wedge Val(w_{turn}(c)) = \mathsf{true} \wedge P(w_{turn}(c)))$. This sentence in L_{BP} also explains why we prefer mathematical English over formal first-order sentences when writing for the human eyes.

The list of "basic properties" (stated in this language L_{BP}) is in Figure 3. It is divided into four parts: the general properties, the register properties, the P properties, and the Q properties.

We ask the reader to reprove Theorem 3.1 and to check all details in order to make sure that all assumptions that are needed for the proof are stated in this list of Basic Properties.

4 A formal framework based on local states

In the previous section we took the basic properties of Figure 3 as self-evident and showed that they imply the mutual exclusion property (Theorem 3.1). How can we formally prove that executions of the Peterson algorithm satisfy these basic properties? For this aim, we must be able to view the executions of the algorithm as first order structures, that is structures for which the truth of first order formulas can be evaluated (we call such structures *Tarskian*). Traditional and well-established methods that view histories of global states as models of some temporal logic language will not work because the statements of the basic properties involve quantification over events and the introduction of functions from the events into the events (such as the $sr(c)$ function). Our aim here is to explicate the meaning of the following theorem and to outline a proof.

Theorem 4.1. *Every execution of the algorithm of Peterson satisfies the properties of Figure 3.*

There is more than one way to define the term "algorithm execution" that appears in the theorem and to prove it. It is possible to take a history of global states and to expand it into a Tarskian structure by adding the step occurrences as the events of the structure. The resulting structure, if presented as a Tarskian structure, will satisfy these properties. But for reasons that will be discussed in

general properties: The Event type is partitioned into P and Q events (predicates P and Q are called "processes"). Relation $<$ is an irreflexive and transitive relation on Event. The restriction of $<$ to each of the two processes is a linear ordering.

register properties: The function *register* gives for every *read* and *write* event e a register name $register(e) \in \{\mathsf{turn}, \mathsf{interested}_P, \mathsf{interested}_Q\}$. If $write(e)$ ($read(e)$) and $register(e) = \mathsf{R}$ (one of the three registers) we say that e is a write event (read event) in register R. We say that $Val(e)$ is its value.

For every register R, the set of *read*/*write* events is linearly ordered by $<$.

Every register has an initial write event that precedes any non-initial events. The value of the initial write events on $\mathsf{interested}_P$ and $\mathsf{interested}_Q$ is false.

If r is a read event of register R, and if $w < r$ is a write event and $Val(r) \neq Val(w)$ then there is a write in R event w' so that $w < w' < r$.

P **properties:** For every cs event c by process P, $w_{interested}(c)$ is a write event by P in register $\mathsf{interested}_P$ of the value true. $w_{turn}(c)$ is a write event by P in register *turn* of the value Q. $sr(c)$ is a read event by program P.

$$w_{interested}(c) < w_{turn}(c) < sr(c) < c. \tag{2}$$

There are two cases.

1. $register(sr(c)) = \mathsf{interested}_Q$ and $Val(sr(c)) = \mathsf{false}$.
2. $register(sr(c)) = \mathsf{turn}$ and $Val(sr(c)) = \mathsf{P}$.

If w is any write event by process P such that $w_{interested}(c) < w < c$ then $w = w_{turn}(c)$.

Any write by process P is either in register *turn* or in $\mathsf{interested}_P$. Any write in *turn* by process P is of value Q.

Q **properties:** These are obtained as above by interchanging P and Q.

Figure 3: The Basic Properties, stated in the L_{BP} language.

Section 5 we prefer to consider local histories rather than histories of global states. The main idea that leads the proof of this theorem, as we see it, is quite simple, but its fuller development will take more space then a short article allows. I believe that if I describe this idea in general lines, then it would be easier for the reader to follow the (partial) details that follow, and in fact some readers may prefer to develop their own versions of this proof.

Recall that the list of Basic Properties of Figure 3 is divided into general properties, register properties, P properties, and Q properties. The main idea in our proof is a *separation of concerns*: The register properties are not connected to any algorithm and they stand independently as an *assumption* that we make on the registers. The P properties of that list depend only on the code of the P protocol. We could deduce them from the code of P even if we do not know that there is another process named Q that executes concurrently. Hence the list of P properties can be deduced by considering only local states and steps that refer to local variables of P and their values. In a similar vein the Q properties depend only on the Q protocol. Now suppose that we have a structure (a system execution) in which both events by process P and events by process Q appear, and the partial precedence relation on these events is not necessarily linear. Two *projections* can be defined, one which takes only the events of P and forms a substructure that consists only of these events; let's call it M_P. And another projection yields the structure M_Q obtained by collecting only events of Q. There are basically three requirements on M: 1. that the read/write events satisfy the specification of serial registers. This is an assumption that we make; 2. that the structure M_P satisfies the P properties of Figure 3; and 3. that the M_Q structure satisfies the Q properties. The fact that M_P and M_Q satisfy these properties is a consequence of the assumption that these structures are derived from local histories of processes P and Q respectively.

Now that we understand the role of local histories in our proof, we shall define in detail the local variables, states, steps, and histories of process P. The corresponding definitions for Q can be defined in the same manner. The *local variables* of P are $V_P = \{PC_P, a, b\}$ (we should write a_P and b_P to distinguish these local variables from the corresponding Q variables, but the context makes it clear that we refer here to local P variables). The type of PC_P (its set of possible values) is the set of control positions which we take to be the line numbers $\{1_P, \ldots, 6_P\}$ (which include $4_P(1)$ and $4_P(2)$). The type of a is the set of boolean values $\{\mathsf{true}, \mathsf{false}\}$ and the type of b is $\{\mathsf{P}, \mathsf{Q}\}$. Note that the registers are *not* local variables. In general, the communication devices (queues and registers for example), which are variables of global states, are not variables of local states in the behavioral approach.

A *local state* of P is a function S that is defined over V_P and gives to every variable a value in its type. A state is also viewed as a structure.

An initial local state for P is one in which the value of PC_P is 1_P.

Local steps of P are pairs of local states (S, T) that describe an execution of an instruction in the code of P. We have the following local P steps. $(1_P, 1_P)$, $(1_P, 2_P)$, $(2_P, 3_P)$, $(3_P, 4_P)$, $(4, 4)$, $(4, 5)$, $(5, 6)$, $(6, 1)$. With every local step we may attach some predicate and some function values. We give a few examples.

A $(2_P, 3_P)$ step is a write in register $\mathsf{interested}_P$ of the value true, this means that the *write* predicate is attached to this step and its value is true, and the *register* function which tells us in which register this step acts has the value $\mathsf{interested}_P$. However, this step does not record the value of the write in any register variable, simply because registers are not local state variables. Formally, a $(2_P, 3_P)$ step is a pair of local P states (S, T) so that $S(PC_P) = 2_P$, $T(PC_P) = 3_P$ and for any other local P variable v $S(v) = T(v)$.

A $(3_P, 4_P)$ step is a pair of P states (S, T) so that $S(PC_P) = 3_P$ and $T(PC_P) = 4_P$, $T(a) = \mathsf{true}$, and $T(b) = \mathsf{Q}$. We characterize this step as a write in register *turn* of the value Q. But again, unlike the case of global steps in the assertional method, this write value is not recorded in any variable.

A $(4_P, 4_P(1))$ step is a pair of local P states so that $S \models a \wedge (b \neq \mathsf{P})$, $S(PC_P) = 4_P$, $T(PC_P) = 4_P(1)$, and no other variable changes. (The role of this step is to indicate that register $\mathsf{interested}_Q$ is going to be read next.)

A $(4_P(1), 4_P)$ step is a pair of local P states (S, T) with $S(PC_P) = 4_P(1)$, $T(PC_P) = 4_P$ and so that no variable except perhaps a may change its value. This step is predicated as a read step of register $\mathsf{interested}_Q$ and it value is $T(a)$ which may be true or false.

A $(4_P, 5_P)$ step is a pair of local P states (S, T) with $S(PC_P) = 4_P$, $T(PC_P) = 5_P$ and so that $S \models \neg(a \wedge (b \neq \mathsf{P}))$. No other variable changes its value. We say that this step is a cs entry step.

A $(5_P, 6_P)$ step is a pair of local P states (S, T) with $S(PC_P) = 5_P$ and $T(PC_P) = 6_P$, and no other change. This is a cs step. Note that we do no have $(5_P, 5_P)$ steps. The fact that the cs step is a single transition does not indicate that this is an instantaneous event.

A *local history* of P is a sequence of local states of P, $H = (S_0, S_1, \ldots)$ (which we assume to be infinite), so that S_0 is an initial P state and for every i the pair (S_i, S_{i+1}) is a local P step. A local history is not a Tarskian structure, but it can easily be turned into one as we show next.

Given a local P history H, its "history structure", $M = M_H$, can be obtained, roughly speaking, by taking the step occurrences as its sort of events and any other feature of H that is not an event is taken into the atemporal sort of M. The idea is that all the information that we have about H is transformed into M. The set of states and their variables, as well as the functional relation that gives to any state and variable the value of the variable in that state are all incorporated into M.

In order to define M in detail we want first to define the language L_{HP} that M interprets (which speaks about the States and histories of P) and we also have

the corresponding L_{HQ} language for history structures of process Q). L_{HP} extends the language L_{BP} of basic properties. L_{HP} is a two-sorted language with Event and Atemporal sorts, and the Atemporal sort is further divided into the following subsorts: Variables, Variable–values, and States. The variables are $V = \{PC_P, a, b\}$ and the Variable–values are the types of the variables. The type of PC_P are the line numbers $\{1_P, \ldots, 6_P\}$, the type of a is $\{\mathsf{true}, \mathsf{false}\}$ and the type of b is $\{\mathsf{P}, \mathsf{Q}\}$. L_{HP} has the following predicates and functions.

1. $at : States \times V \rightarrow$ Variable–values. (The role of this function, at, is to give for every state S and variable v the value $at(S, v)$ of v at state S.)

2. $pre, post : Event \rightarrow States$. (The role of these functions is to give for every event e the state $pre(e)$ just before action e and the state $post(e)$ which is the result of e.)

3. All predicates and functions of the L_{BP} language are also in L_{HP}.

Suppose that H is a local P history as defined above. The history structure $M = M_H$ that we define now is an interpretation of the language L_{HP}. We define M as a two-sorted structure with universe that consists of events and atemporal sorts defined as follows:

1. The set of natural numbers ω is the sort of *events* of M. (We think of i as the event that caused the transition (S_i, S_{i+1}).

2. The set of local states of P forms the *States* sort of M (an abstract set devoid of any internal structure).

3. The set V of local P variables is also a sub-sort of the atemporal members of M.

4. The set of state variable values is $\{1_P, \ldots, 6_P, \mathsf{P}, \mathsf{Q}, \mathsf{true}, \mathsf{false}\}$.

The structure M interprets the L_{HP} predicates and functions as follows.

1. $<$ is the natural ordering on the set of events ω (a transitive and irreflexive relation).

2. The function $at : States \times V_P \rightarrow$ Variable–values evaluates the states. For every "state" S in M, and "variable" v in V_P, $at(S, v)$ is the value of S at v (what would usually be written as $S(v)$). (Why "state" and "variable" are in quotation marks? Because all members of a structure are devoid of any internal composition, they are plain "points". So S is not really a state, it is not a function; only be means of the at function we can view it as a representation of a state.)

3. Every event $i \in \omega$ is associated with two states $pre(i)$ and $post(i)$. Since M is the structure obtained from the given history H, $pre(i) = S_i$ and $post(i) = S_{i+1}$. Intuitively, $pre(i)$ is the state that event i changes (if there is a change) to S_{i+1}.

4. Predicates *read*, *write*, *cs*, and functions *Register* and *Val* are defined on the *Event* sort. If (S_m, S_{m+1}) is a (i, j) step, then the predicates and function values that were associated to this step apply to event m. We give only some examples in order to illustrate the procedure.

   (a) Define $cs(m)$ when (S_m, S_{m+1}) is a $(5_P, 6_P)$ step.

   (b) If (S_m, S_{m+1}) is a $(2_P, 3_P)$ step then we define $write(m)$, $register(m) = \mathsf{interested}_P$, and $Val(m) = \mathsf{true}$.

   (c) If (S_m, S_{m+1}) is a $(4_P(1), 4_P)$ step then we define $read(m)$, $register(m) = \mathsf{interested}_Q$, and $Val(m) = S_{m+1}(a)$.

We define $\mathcal{S}_P = \{M_H \mid H \text{ is a local history of } P\}$. So $\mathcal{S}_P$ is the system of all possible local history structures of P. If α is any sentence in the L_{HP} language (and in particular if α is in the L_{BP} language and it speaks only about events in P), then the statement "every local history of P satisfies α" is explicated as "if $M \in \mathcal{S}_P$ then $M \models \alpha$". In order to formally prove such statements, we shall define a sentence $\beta = \beta_P$ in the L_{HP} language whose ω models are just the structures of $\mathcal{S}_P$. The program sentence β says that its models are indeed executions of the algorithm of P. (If we are ready to use second order sentences then β can say that the ordering $<$ on the *Event* has the property that every nonempty set has a first element, and so it can determine that the *Event* type is isomorphic to the natural numbers, but this is not necessary.) In addition to determining the properties of the ordering $<$, β is obtained as a conjunction of the following.

1. $pre(0)$ is an initial state.

2. For every i, $pre(i + 1) = post(i)$.

3. For every i, $(pre(i), post(i))$ is a local P step.

Writing out in detail β takes a lot of work, but should not offer any conceptual difficulties. Here I limit myself to providing just one example. To say that sort *States* indeed contains all the functions from the variables into their sorts is a quantified sentence which says that for every choice of values for the variables there is a state that has in fact these values. More formally, this sentence is

$$\forall v_1 \in \{1_P, \ldots 6_P\} \forall v_2 \in \{\mathsf{true}, \mathsf{false}\} \forall v_3 \in \{\mathsf{P}, \mathsf{Q}\}$$
$$\exists S \in States\, (at(S, PC_P) = v_1 \wedge at(S, a) = v_2 \wedge at(S, b) = v_3). \qquad (3)$$

An ω L_{HP} structure is an interpretation of the language in which the sort of events in their $<$ ordering is isomorphic to the set of natural numbers. Clearly, $\mathcal{S}_P$ is the set of all ω L_{HP} structures M that satisfy β. So, if we want to formally prove that a certain sentence α holds in every $M_H \in \mathcal{S}_P$ it suffices to prove that $\beta \vdash \alpha$.

Definable functions. If $\varphi(x, y)$ is a formula such that the program sentence β implies that for every event x there exists one and only one event y so that $\varphi(x, y)$, then we may introduce a new function symbol f_φ and use it in our formulas, after adding the defining axiom to β: $(y = f_\varphi(x)) \equiv \varphi(x, y)$. Here is an example. Let $\varphi(x, y)$ be the following formula in the L_{HP} language.

> If x is a cs event then y is a read event, $y < x$, there is no write event w with $y < w < x$, and one of the following possibilities holds:
>
> 1. y is a read of register $\mathsf{interested}_Q$ with value false, or
> 2. y is a read of *turn* with value P.
>
> If x is not a cs event then $y = \perp$.

It follows from β that for every event x there is one and only one y so that $\varphi(x, y)$. The proof may be long, but it is not difficult and it enables the definition of the function $sr(x)$ and then the proof of its properties. Similarly, the functions $w_{interested}$ and w_{turn} can be defined and the third part from the list of Basic Properties of Figure 3, namely the Basic P properties, can be proved to follow from β_P. In a similar way the Basic Q properties follow from β_Q.

Let GP be the general properties that are listed in Figure 3. These are assumptions that we make on the precedence relation $<$. Let RP be the register properties that are listed in Figure 3. (Namely that for every register R, the set of *read/write* events is linearly ordered by $<$, and that if r is a read event of register R, and if $w < r$ is a write event and $Val(r) \neq Val(w)$ then there is a write in R event w' so that $w < w' < r$.) These RP statements are assumptions that we make on our communication devices.

It follows that the list of basic properties of Figure 3 can be formally deduced from the conjunction $GP \wedge RP \wedge \beta_P \wedge \beta_Q$. We formulate this theorem for future reference.

Theorem 4.2. *Statement $GP \wedge RP \wedge \beta_P \wedge \beta_Q$ implies the Basic Properties of Figure 3.*

5 A short discussion

I hope that my reader was satisfied with the behavioral proof in section 3 in which the mutual exclusion property was deduced from the Basic Properties. That proof

not only established the mutual exclusion property but actually showed how the writes in the *turn* register determine the order in which the critical section events are executed. I am not so sure about Section 4, in which a direction is indicated how to prove that the basic properties indeed hold in every execution of the algorithm. I guess that my reader was left with many doubts and questions. "Why you shunt global states?" she may ask. "Would it not be possible to employ global states (with which we are more familiar) and histories of global states in order to define executions of the algorithms that satisfy the Basic Properties of Figure 3?" That's true, it is possible to transform a history of global states into a Tarskian structure that supports the satisfaction relation for first order languages. This would not be very different from what we did in Section 4, and the main idea would be to take the step occurrences as elements of the resulting structure. But there are three main (related) reasons why I prefer to base the proof on local states of each of the processes. The Ockham razor principle advises us to use the simplest possible tools, and since local histories of sequential processes are conceptually simpler than interleaving histories with global states, they are preferable. There is another reason. If we use global states and histories of global states, then the registers, being variables of the global states, determine the relations between the read and write events in an obvious way. And then properties like the *RP* properties of Figure 3 follow as theorems that we have to prove about the resulting history structures. But conceptually, the register properties are assumptions made on the communication devices (serial shared memory registers in this case). That is, the theorem that we prove (Theorem 4.2) has the form: *if the read/write actions in the registers behave as required by their specifications and if the processes execute their algorithms, then the correctness property (mutual exclusion in our case) holds.* So, it seems more appropriate to have the register properties as assumptions than as consequences of some modeling paradigm.

Finally, it is our experience that the pattern of behavioral proofs exemplified here in this extremely simple critical-section algorithm of Peterson is quite general. Given a distributed algorithm for serial processes $P_1, \ldots, P_N$ that execute concurrently, in order to prove a correctness statement α (such as the mutual exclusion property) one formulates some basic properties BP_i that are properties of P_i's executions alone without any reference to other processes and can be proved by reference to local states and histories of P_i. Then one assumes the properties of the communication devices, CP, which may be shared memory (for example regular registers) or message channels etc. And then α is shown to be a consequence of the conjunction of CP and all the basic properties of the processes. Following [2] in which the usage of Tarskian system executions (or event structures) was promoted, we say that the "unrestricted" semantics of a distributed system is the description of the system by local states and histories of the participating processes. The unrestricted semantics is the description of the system when the

communication devices operate in a random way with no relation between the different actions on the communication devices (such as read and write, or send and receive events). In a slogan we can say that the correctness of a distributed system is a consequence of its unrestricted semantics together with the specification of the communication devices. In the assertional approach the semantics of the programs and communication devices are all interwoven within the notion of global states, but in the behavioral approach there is a separation of issues; the semantics of each process is done with local histories of that process and the semantics of the communication devices is expressed without any connection to a specific algorithm.

References

[1] U. Abraham. On Interprocess Communication and the Implementation of Multi-Writer Atomic Registers. Theor. Comput. Sci. 149(2): 257-298 (1995).

[2] U. Abraham. *Models for Concurrency*. Gordon and Breach, (1999).

[3] U. Abraham. Logical Classification of Distributed Algorithms (Bakery Algorithms as an example). Theor. Comput. Sci. 412(25): 2724-2745 (2011).

[4] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.

[5] G. L. Peterson. Myths about the mutual exclusion problem. Inf. Process Lett. 12, 3 (June) 1981, 1133-1145.

[6] A. U. Shankar, An Introduction to Assertional Reasoning, ACM Comput. Surv.(1993) 225-262.