

# Self Stabilizing weak snapshots and timestamps

Uri Abraham

Departments of Mathematics and Computer Science  
Ben-Gurion University, Beer Sheva, Israel

September 16, 2007

## Abstract

The first part of this paper contains a variant of the weak snapshot algorithm of Dwork, Herlihy, Plotkin, and Waarts [17] and proves an additional property not considered by its authors, namely its self-stabilization. In the second part we employ that weak snapshot and combine it with our weak timestamp algorithm of [5] in order to give a full fledge self-stabilizing timestamp algorithm with linear time complexity (in the number of processes). We show how predicate calculus and structures can be employed to formally specify self-stabilization and to prove that the algorithm self-stabilizes.

## 1 Introduction

Timestamps are used by shared-memory multiprocessors in a great variety of ways, and they appear in many forms (dates, number of milliseconds since the system started etc.). A reasonable way to unify these various appearances is to think of timestamps as natural numbers attached to data values read and written by the processes. These numbers allow the readers to know the order in which the writes were executed. Of course, timestamps are equally useful for message passing communication, but in this paper we deal exclusively with shared-memory communication in which single-writer multiple reader registers are used by the processes. An interesting question which has

attracted some attention and prompt the development of quite sophisticated algorithms is that of obtaining timestamps with only a bounded number of values, necessarily requiring that the same values are reused time and again in any infinite execution. The first works in which this question was dealt with were Israeli and Li [19] (sequential timestamp systems) and Dolev and Shavit [14] (concurrent timestamps). These papers contain some of the basic ideas that are used again in subsequent works on bounded timestamps, and most importantly they provide a formal definition of the problem—an abstract definition of timestamp systems—which is a necessary prerequisite for any bounded solution.

Already Dolev and Shavit [14] saw the connection between the bounded timestamp problem and the atomic snapshot scan operations. This connection was further exploited by [17]: they defined a notion of *weak snapshot*, showed how to efficiently implement weak snapshot operations (scan and update), and used these operations in the Dolev–Shavit algorithm in order to obtain an  $O(N)$  efficient bounded timestamp algorithm (with  $O(N)$  sized registers).

Self-stabilization (to define it in one sentence) is the paradigm that seeks resilient algorithms that will eventually return to normal operation even under arbitrarily chosen initial conditions. For self-stabilizing timestamps, the requirement that the values are bounded is essential, and the reason is the following. Unbounded timestamps are never really unbounded in applications, since all the registers used by the processes are of fixed bounded size. The argument put forward for viewing these bounded registers as representing unbounded numbers is that even for moderately sized registers there is an exponential store of possible values which will never be consumed in the life-time of any perishable machine. This valid argument will not work for self-stabilization, since that paradigm assumes that any initial value is possible, including one in which the register reaches its “last” value. Hence the self-stabilizing timestamp problem calls for bounded valued timestamps.

The problem of devising a self-stabilizing timestamp system is solved (for the first time) in [5], but the self-stabilizing timestamp algorithm presented there is rather inefficient since each scan operation requires  $2N^2$  reads of single writer registers (where  $N$  is the number of processes), and a natural question is whether a self-stabilizing timestamp with only  $O(N)$  read/write register accesses per scan and update operations is possible. Our main mission in this paper is to give a positive answer, thereby showing that a self-stabilizing timestamp algorithm can be almost as efficient as the best bounded times-

tamp available today: that of Dwork et al. [17]. The size of our registers is  $O(N \log N)$  (bits per register) while [17] use registers of size  $O(N)$ . So there is still some improvement to be made and the following open question remains: is there a self-stabilizing timestamp algorithm in which each operation takes  $O(N)$  read/write events on registers with  $O(N)$  bits.

At this point we can describe our present work as an observation made after reading [17]<sup>1</sup>, namely that their basic idea is also applicable to our self-stabilizing algorithm of [5], and its computational complexity can be reduced by simply replacing blocks of the algorithm with weak snapshot operations. Our paper is an elaboration on this observation, and we demonstrate here an implementation of bounded self-stabilizing timestamps with operations that take  $O(N)$  read/write accesses to registers of  $O(N \log N)$  bits.

Our paper is considerably long, and one may suggest to shorten it by relying on the results of [17] and using their algorithm as a block in our stable timestamping algorithm. This would not save much work, however, and we prefer to develop our version of [17] from scratch mainly for the following reason. We need for our application not just any weak snapshot algorithm, but a self-stabilizing one in order to finally obtain a self-stabilizing timestamp algorithm. We do not know whether the algorithm of [17] is self-stabilizing or not, but, even if it were, the proof of that fact would involve a complete presentation of the algorithm with a correctness proof written anew, which would disallow any potential saving. Moreover, we cannot substitute any weak snapshot registers in our algorithm, since we require additional properties (the “axioms” of section [?]).

The weak snapshot algorithm of [17] is presented in that paper in a modular way: first an unbounded algorithm is given, and then, relying on the Traceable Use abstraction of Dwork and Waarts ([16]) the unbounded weak snapshot is converted into a bounded one. The correctness proof of [17] is done indirectly by reducing it to the correctness of the Dolev-Shavit timestamp algorithm [14], and it is unclear to us whether this reduction can serve as a basis for a self-stabilization proof. We offer here an algorithm which, we believe, is based on different ideas than those of [17] and which may prove to be useful for other purposes. Our correctness proof is complete and self-contained, and thence the length of our paper.

---

<sup>1</sup>[17] was published before our [5], however we were not aware of their results when [5] was written (and certainly not in 1990, when our bounded timestamp algorithm was first described in an unpublished paper).

We have here an additional aim which should be mentioned since it is as important in our view as the self-stabilizing algorithms presented here. A large part of our work is dedicated to a presentation of a formal framework suitable for specification and correctness proofs of protocols, and to a justification of its premisses. In particular, our definition and treatment of self-stabilization is different from the usual one, since we are using predicate calculus rather than the notion of global states (in this, we follow [7] and [5]). This necessitates a rather unusually long introductory material which, we hope, is valuable for its own sake.

## 2 Time-lapse snapshots

The timestamp algorithm presented in the second part of this paper employs weak snapshots as abstract building blocks. That is, the algorithm contains update and scan instructions whose implementation is left unspecified: it may be the weak snapshot algorithm presented in the first part of this paper, but it could also be any other appropriate implementation. The correctness proof of the timestamp algorithm depends on abstract properties of the weak snapshot actions, and any snapshot algorithm that satisfies these specifications can be used. Specifically, for the correctness of our timestamp protocol, we require update/scan actions that satisfy the intermediate axioms of Section 3. We also prove that these axioms imply the weak snapshot properties. Since the scan/update algorithm of the first part of our paper is proved to satisfy the intermediate axioms, it follows that it possesses the weak snapshot properties as well. In order to formally prove this implication, namely that

$$\text{intermediate axioms of section 3} \Rightarrow \text{weak snapshot properties} \quad (1)$$

we must describe a conceptual framework in which such an implication can be conducted. This is a major issue in our opinion, to which a large part of the paper is dedicated. Any such framework comprises two related parts syntax and semantics:

1. A formal language  $L$  has to be fashioned in which both the axioms of section 3 and the weak snapshot properties can be expressed. So the informal statement 1 is rendered as a sentence  $\varphi \rightarrow \psi$  in that language, where  $\varphi$  expresses the axioms of section 3 and  $\psi$  reflects the weak snapshot properties of [17].

2. A collection of structures has to be defined which interpret  $L$ . For every such structure  $M$  and formula  $\alpha$  in  $L$  the satisfaction relation has to be defined, which says that  $\alpha$  holds true in  $M$  (under a possible assignment of values to its free variables).

Then a proof that  $\varphi$  implies  $\psi$  amounts to proving that it is logically valid:  $\varphi \rightarrow \psi$  holds in any interpretation of  $L$ . (This is a “model theoretic” proof, one which relies on the notion of logical implication understood in terms of structures, rather than a formal calculus based on formal rules of deduction.) What should be the character of the language and structures of an appropriate framework in which an implication such as (1) can be proved correct? Many investigators of concurrency have opted for temporal language and history structures (sequences of states) as their modelling framework. See, for example, Lamport’s Temporal Logic of Actions [23] or Manna and Pnueli’s textbook [24]. This paper takes a different approach and suggests predicate language (mostly first order) and its structures (Tarskian structures) for modelling concurrency. The choice of models (language and structures) is matter of art and experience, researchers may have different preferences, and it is conceivable that for some applications one approach is better suited whereas for other applications another approach is more appropriate. Assembling a suitable framework has to be done with great care in order to address the faithfulness issue which is the following: when defining items 1 and 2 above, we must argue that our formula  $\psi$ , for example, is an adequate and faithful rendering of the original notion described informally (or semi-formally in [17]). This is a rather general problem of course, not specific to the question of weak snapshots, and we make the following note (see [8] for a longer discussion of this issue) .

We distinguish between two ways to formalize a given text and informal specification . The first, which we call *translation*, is a formal specification in a language that bears little resemblance to the original informal specification, and is orthogonal to its conceptual world. The resulting translation is formal indeed, and can even be an accurate definition of the object formalized, but it may also only partially and indirectly reflect the intuitions of the original text. The second way, which we call *explication*, requires a deeper analysis of the original text in order to find out the hidden assumptions and partially pronounced concepts that its authors are using. The resulting formal explication is closer in spirit to the original text, and (ideally) reflects its intensions and intuitions.

We aim here for a formal explication, rather than translation, of the weak snapshot specification of [17], and for that reason we first report from that paper and describe their model and definitions of snapshots and weak snapshots (namely section 2 of that paper). In the remaining subsections we formalize these notions using quantifier language and Tarskian structures.

## 2.1 A report of section 2 of [17]

We read the following in Section 2 of [17]. A *concurrent system* consists of a collection of  $N$  asynchronous *processes* that communicate through an initialized shared memory. The memory is an array of  $N$  locations, called registers. Each memory register can be written by one “owner” process and read by any process (*single-writer multiple-reader registers*). Reads and writes to shared registers are assumed to be *atomic*, that is, they can be viewed as occurring at a single instant of time. In [17] an execution of a protocol is viewed as an interleaving of atomic reads and writes actions by the different processes. The collection of all atomic read/write actions is linearly ordered (and the order type is either that of the natural numbers or a finite initial segment).

An *atomic snapshot memory* is defined in [17] as a system that supports two kinds of abstract operations: **update** modifies a location in the shared array, and **scan** instantaneously reads (makes a copy of) the entire array. Formally, the definition relies on the following notation:

Let  $U_i^k$  denotes the  $k$ th **update** operation of process  $i$ , and  $S_i^k$  the  $k$ th **scan** operation of process  $i$ .

A relation, called *precedes* and denoted  $\rightarrow$ , is defined on the operations. Operation  $A$  *precedes* operation  $B$ , written as  $A \rightarrow B$ , if  $B$  starts after  $A$  finishes. In details, each operation of process  $i$  consists of a (usually finite) number of read/write atomic actions on the registers, and  $A \rightarrow B$  is defined if and only if the last action in  $A$  precedes the first action of  $B$  (in the assumed linear ordering of all atomic actions). We assume that the operations of each individual process  $i$  are linearly ordered by  $\rightarrow$ . Now the requirements for atomic snapshot memory can be defined:

There exists a total order “ $\Rightarrow$ ” on operations such that

1. If  $A \rightarrow B$  then  $A \Rightarrow B$ .

2. If **scan**  $S_p$  returns  $v = \langle v_1, \dots, v_N \rangle$ , then  $v_q$  is the value written by the latest **update**  $U_q$  by process  $q$  ordered before  $S_p$  by  $\Rightarrow$ .

Finally, a weak snapshot memory system (synonymously called *time-lapse snapshot*) is defined by repeating the definition given above for atomic snapshot, but allowing  $\Rightarrow$  to be a partial (irreflexive) ordering of the **scan/update** operations (instead of the linear ordering required for atomic snapshots).

An important remark made by [17] is that this definition of weak timestamps makes sense, since for each process  $q$  the **update** operations by  $q$  are linearly ordered (and there is an initial one that precedes any **scan** operation) so that “the latest **update**  $U_q$  by process  $q$  that precedes  $S_p$ ” is a well defined object.

In [17] it is mentioned that the work of [21] has influenced the lapse-time snapshot concept and that, if the scanner is a serial process then a weak snapshot is also single-reader composite register for the case in which there is a single writer per variable (to prove this, we employ Theorem 1 in [21]).

## 2.2 The need for Explication

From a logical point of view, something is missing from the model and definitions of atomic and weak snapshots reported above. The aims of [17] include the presentation of a new algorithmic idea, that of weak timestamps, to implement it and to show its usefulness. For these aims, the paper is certainly complete. But, as we argued above, there is a need to explicate the foundational framework and assumptions that underlie [17]. From a modelling point of view, that the definitions of atomic and weak snapshots reported above lack the support of a formal language and its interpretations. There is no doubt that an expert could translate the definitions and proofs given in [17] into TLA+, or any of the other state-based formalisms available, and obtain a completely formal modelling and proofs—this was done for more complex algorithms. But this would not be an explication of [17]. By explication we mean a formalization that is faithful to the text and its spirit. We would like to inquire about the modelling tenets which the text of [17] seems to subscribe to. We shall claim that it is not toward temporal logic, but rather to the Tarskian paradigm of predicate logic and structures that the text is inclined. Since [17] does not describe its formal modelling choice, we have to deduce it from the text, to supplement details as they become necessary, and

to interpret the text in ways which may reflect our tendencies rather than that of its authors.

The language of temporal logic is characterized by its use of the modal operators  $\Box$  and  $\diamond$ , corresponding to “Henceforth” and “Eventually”, and its structures are characterized by the presence of states and transitions. In [17], however, we find no temporal expressions, no states seem to be mentioned, and no transitions are used in the specifications. It seems that an explication of [17] should not rely on temporal logic if it wants to reflect the paper’s informal paradigms.

The first question that we ask in our analysis is this: What are the basic objects that populate the universe of discourse of [17]? These are clearly the events, and they come in two sorts: the atomic actions (read and write) and the operations (**update** and **scan**). Events of the first sort will also be called here “low level events” and the operation will also be called “high level events”. Any high level event is a set of lower level events. Thus expressions of [17] that refer to the first and last actions of an operation make sense: “Each **scan** and **update** operation takes place during the interval of time beginning with the first atomic action of the operation and ending with the last atomic action.” We can gather from this quotation that time intervals too are members of the universe of [17]. It is true that “events” appear also in the temporal logic paradigm: transitions are often called events and indeed they represent events. But these are the low level events; higher level events do not appear in temporal logic structures—not as prime objects. There are other basic objects in [17] besides the events. These are the values that are written (by update operations) and arrays of values returned by scan operations. In addition, the processes or the processes’ indexes appear as basic objects. For example, for a process  $a$  we find “Let  $a$  perform an **update**  $U_a$ , writing the value  $v_a$ .” When we see a sentence of the form “ $v$  is the value written by **update**  $U$ ” we can think of it as a binary predicate,  $\text{ValueWritten}(v, U)$ , which relates a value  $v$  to an event  $U$ . The article “the” in the above quotation implies, however, that each **update** has a single value to which it corresponds, and so perhaps it is more appropriate to render that sentence as a functional relation  $v = \text{Value}(U)$ , thereby viewing  $\text{Value}$  as a function from the set of operations into the set of values. Besides functions, the text employs relations, for example the  $A \rightarrow B$  relation or the partial linearization relation  $A \Rightarrow B$ . We note that in temporal logic there are no relations or functions on the events, and this strengthens our claim that temporal logic should not be used for explication of [17].

The status of the symbols  $U_i^k$  and  $S_i^k$  was not immediately clear to me. These appear, for example, in the expression “Let  $U_i^k$  denotes the  $k$ th **update** operation of process  $i$ ”, and similarly we read that  $S_i^k$  is used to denote the  $k$ th **scan** operation of process  $i$ . The meaning is clear, but I could not resolve if  $U_i^k$  is a single symbol or a compound expression composed of  $U$ ,  $i$ , and  $k$  which are separate entities of the language. If it is a single symbol, then we note that it serves in [17] both as a variable which can be quantified, a predicate which says that this operation is by the  $i$ th process, and another predicate which says that this operation is the  $k$ th operation of that process. The tradition in mathematical logic is to have a specific and restricted usage for each symbol, but computer science parlance is much more flexible, and overloading of symbols is very common. This overloading can be useful, but sometimes it leads to cumbersome formulations. For example consider the following sentence taken from [17] (monotonicity of scans).

If  $S_a^i$  and  $S_b^j$  are two **scans** satisfying  $S_a^i \rightarrow S_b^j$  ( $a$  and  $b$  could be the same process), and if  $S_a^i$  observes **update**  $U_c^k$  (formally,  $U_c^k \Rightarrow S_a^i$ ), then  $S_b^j$  observes  $U_c^k$ .

We see here how  $S_a^i$  and  $S_b^j$  are used as quantified variables: that sentence says “for every  $S_a^i$  and  $S_b^j$ , scan operations by  $a$  and  $b$  that are, respectively, the  $i$ th and  $j$ th operations that satisfy  $S_a^i \rightarrow S_b^j$  etc. It seems that a clearer formulation of the same property would be:

If  $S$  and  $S'$  are any **scan** operations (by the same or different processes) such that  $S \rightarrow S'$ , and if  $U$  is any **update** operation such that  $U \Rightarrow S$ , then  $U \Rightarrow S'$  as well.

Perhaps we will be more faithful to the intentions of the text by understanding  $U_a^k$  as a functional expression. That is,  $U$  is a function from the product  $\mathcal{N} \times \{1, \dots, N\}$  into the set of operation executions, and for every natural number  $k$  and index  $a$ ,  $U_a^k$  is another way of writing  $U(k, a)$  (namely the application of  $U$  to the pair  $(k, a)$ ). Then the universal statement (appearing in [17]) “If  $U_a^i$  and  $U_b^j$  are two **update** operations etc.” should be understood as “for every processes  $a$  and  $b$ , and natural numbers  $i$  and  $j$  etc.” This usage makes perfect sense, but we shall not adopt it here in our formalization because we prefer to make do without the natural numbers. (Admittedly, this is a stylistic choice since we do have the sort of moments.) Already at this point one may be convinced that the language employed in [17] is not

temporal logic, but rather some flexible variant of a first-order language that involves low and high level events, values, predicates and functions. This reinforces the feeling that the structures tacitly assumed by [17] are Tarskian structures, the structures that we will suggest here to explicate the modelling framework. Generally speaking, a Tarskian structure  $M$  consists of a universe (denoted  $|M|$ ) and interpretation  $V^M$  for every symbol  $V$  of the signature. When the identity of the structure  $M$  is clear, we may omit the superscript  $M$  and use the symbols to refer to their interpretations. So we can use  $\rightarrow$  instead of  $\rightarrow^M$  etc.

In the following subsection we define a formal predicate language in which weak snapshots can be specified, and then we will define the possible interpretations (structures) of this language. Why is such a formalization of the discourse language desirable? Can't one continue with an informal technical discourse? One can, of course, and in some occasions it may even be more appropriate than a formal writing, but an informal discourse has its pitfalls and is sometimes harder to understand than a well-built formal or semi-formal description. Moreover, in the second part of this paper we will describe and prove the correctness of a timestamping algorithm that uses the weak **scan** and **update** operations as its primitive communication actions. A formal correctness proof of such an algorithm must rely on formally stated properties of these operations and hence the need for our formal explication.

### 2.3 Tarskian system executions

In order to define a signature for a multi-sorted language one has to make a list of the following items.

1. Names of sorts. The sorts are the types of objects that populate the universes (interpretations) of the signature language. For example, we will have the *Data* and *Event* sorts among the sorts of the *wss* signature presented below. The signature may require that some sort is a subset of another sort (as for example the integers is a sub-sort of the reals in a signature for analysis). It can also require that two sorts are disjoint (for example the *Event* and *Data* sorts).
2. Names of variables. The signature can connect variables to specific sorts. For example, we can stipulate that  $e_0, e_1, \dots$  are all *Event* variables, or that  $m, n, k$  are integer variables. This allows one to form shorter quantified formulas. For example,  $\exists e\varphi(e)$  would mean: there

exists an *Event*  $e$  such that  $\varphi(e)$  holds. It is also possible to use sorts as predicates and to write that statement as  $\exists x(\text{Event}(x) \wedge \varphi(x))$ .

3. The signature lists predicates (name of relations) and for each predicate its arity. The signature can also stipulate the sort of the  $k$ -th entry of a predicate. For example, predicate  $\rightarrow$  has arity 2 (a binary predicate) and both of its arguments are of sort *Event*.
4. The signature lists names of functions, and their arities. The signature can also stipulate the sorts of the function entries and the sort of values taken by the function. For example, we have in our *wss* signature a unary function called *begin* which applies to events and takes values of sort *Moment*. (The intension is that  $\text{begin}(e)$  denotes the moment event  $e$  begins.)
5. The signature lists constants and their sorts.

Given any signature, there is a standard inductive definition of the resulting *language*, namely the set of all expressions and formulas built with quantifiers and connectives from the elements of the signature.

An interpretation  $M$  of a signature consists of:

1. A universe, namely a non-empty set  $A$  of elements.
2. For each sort  $S$  in the signature a set  $S^M \subseteq A$  which represents the members of  $M$  of sort  $S$ .
3. For every relation symbol  $R$  of arity  $k$  in the signature, a relation  $R^M \subseteq A^k$ . In fact, if the signature stipulates that the  $i$ -th entry of  $R$  is of sort  $S_i$  then we have  $R^M \subseteq S_0^M \times \dots \times S_{k-1}^M$ .
4. For every function symbol  $F$  in the signature, of arity  $k$ ,  $F^M$  is a function from  $A^k$  to  $A$ . Again, it has to respect the stipulations made by the signature about the domain and value sorts of the function.
5. For every constant  $c$  of sort  $S$  in the signature,  $c^M \in S^M$ .

The structures that we are going to define next are variants of system executions introduced in [22] in which their status as interpretation of predicate language signatures is explicitly stated (in this we follow [4]).

**Definition 2.1** *A system execution signature is a signature that contains the following items (and possibly more).*

1. There are three sorts: *Event*, *atemporal*, and *Moment* (there may be additional sorts, but these three are required). *Event* and *atemporal* are disjoint, and *Moment* is a subsort of *atemporal*.
2. There is a binary relation symbol  $\rightarrow$  defined on *Event*. It is called the temporal precedence relation. Intuitively  $e_1 \rightarrow e_2$  means that event  $e_1$  finished before  $e_2$  begins. We may write  $e_1 \rightarrow = e_2$  as a shorthand for “ $e_1 \rightarrow e_2$  or  $e_1 = e_2$ ”.
3. There is a binary relation  $<$  on *Moment*. Intuitively,  $m_1 < m_2$  means that moment  $m_1$  precedes (is earlier than) moment  $m_2$ .
4. There are two functions *begin* and *end* from *Event* into *Moment*. We think of event  $e$  as extended in time and represented by the closed interval  $[begin(e), end(e)]$ .
5. There are possibly other predicates, functions, and constants in the signature.

**Definition 2.2** *A system execution is an interpretation  $M$  of a system execution signature so that:*

1.  $Moment^M$  is linearly ordered by  $<^M$  and the order-type is that of the natural numbers.
2. The following holds in  $M$ :
  - (a) For every event  $e$ ,  $begin(e) \leq end(e)$ .
  - (b) For every events  $e_1$  and  $e_2$ ,  $e_1 \rightarrow e_2$  iff  $end(e_1) < begin(e_2)$ .

We note that our definition of system executions is somewhat simplified in tacitly assuming that all events are terminating. There are applications which necessitate non-terminating events—that is with infinite duration. For such events we would have to assume that  $end(e)$  is undefined, or else that it is a special value  $\infty$ .

## 2.4 The weak-snapshot language

We define next a system execution signature (called *wss*) and then a list of properties in the resulting language. The collection of all system executions that satisfy these properties is the definition of weak snapshot behavior. The signature *wss* comprises the following items.

1. There are two main sorts: *Event*, and *atemporal*, and the latter sort contains four pairwise disjoint sorts: *Data*, *DataArray*, *UpdaterIndex* and *Moment*. We use the letter *e* (possibly with indices) and upper case letters as variables over *Event*, and other letters as atemporal variables.
2. There are two unary predicates: **update** and **scan** are applicable on sort *Event*.
3. There are two unary functions: *index* from *Event* into *UpdaterIndex*, and *Value* from *Event* into *atemporal*.
4. There are two binary predicates  $\rightarrow$ , and  $\Rightarrow$  defined on *Event*. ( $\rightarrow$  is the temporal precedence relation symbol.)
5. There is a two-place function symbol  $\Omega$  from  $Event \times UpdaterIndex$  into *Event*.
6. There are the standard functions *begin* and *end*, and the relation  $<$  on *Moment* that exist in any system execution signature.

We will explain the meaning and usage of these symbols and their intuitive meaning in the following notes. An interpretation of the *wss* signature is a structure in which these symbols acquire meaning.

The main sort of objects is *Event*. It is used to represent the high level operations, which are executions of scan and update operations. There is no need to represent the low level atomic actions on the registers when specifying weak snapshot systems because these actions, important as they are for the executions of the algorithm, are not part of the specification. The predicates **scan** and **update** are applicable on *Event*. This means that if *X* is a variable of type *Event* then the formulas **scan**(*X*) and **update**(*X*) can be formed, and given any structure that interprets the *wss* language we expect that for any assignment of *Event* values to *X* such a formula is either true or false. If, however, *d* is an atemporal object (for example in *Data*), then **scan**(*d*) is meaningless.

The *Event* elements are partially ordered by the temporal ordering,  $\rightarrow$ , and it follows that the interpretation of  $(Event, \rightarrow)$  forms a “global-time system execution” in the sense of Lamport’s [22]. We already noted that time and time-intervals appear as essential elements of [17], and we have introduced this feature into our modelling as well. We assume that the sort *Moment* is a set linearly ordered by  $<$  and that its order type is that of the natural numbers. In physics and everyday parlance time is represented as a linear dense ordering (often as real numbers). For the computer-science applications that we have in mind it makes sense to have time as a discrete (rather than dense) ordering. We think of some “global clock” by which the temporal extension of each event can be measured. The processes that execute their programs are not “aware” of *Moment* and have no access to this hypothetical global clock. (Discussions and justification for our representation of time can be found in [6].)

Each event  $e$  is situated in time and it stretches as a closed interval of *Moment* with end points  $begin(e) \leq end(e)$ . (We deal here with terminating events; otherwise we would also have to consider intervals of the form  $[b, \infty)$ .) The temporal ordering  $\rightarrow$  on the events corresponds to the interval ordering:  $e_1 \rightarrow e_2$  iff  $end(e_1) < begin(e_2)$ . Any sort other than *Event* is “atemporal”, namely the temporal ordering  $\rightarrow$  is not applicable to its objects.

Besides *Moment*, we have here, *Data*, *DataArray*, and *UpdaterIndex* as atemporal sorts. *Data* represents the possible values of the memory locations. If  $\mathbf{update}(e)$  then  $Value(e) \in Data$  is the value “written” by  $e$ , and  $i = index(e) \in UpdaterIndex$  is the index of the updater process executing  $e$ . Since every process has its own memory location,  $i$  can also be thought of as representing the memory location on which that process writes. Scan events return *DataArray* values. Any such value is an array indexed by *UpdaterIndex* of *Data* values. This necessitates the introduction of notations like  $a[i]$  to denote the value of array  $a$  at entry  $i$ . There seems to be two alternatives for dealing with arrays in the context of predicate calculus:

1. The first approach is “model theoretic”. If  $a \in DataArray$  and  $i \in UpdaterIndex$ , we must be able to represent the *Data* value of the  $i$ -th entry of  $a$ . The notation  $a[i]$  can be misleading because it seems to imply that  $a$  is a function and  $a[i]$  is the value of this function applied to  $i$ . But  $a$  is not a function—it is a plain member of the universe of  $M$  (a given interpretation of the *uss* signature). This shows the necessity of another three place predicate in our signature  $apply(a, i, d)$  which says

intuitively that  $d \in Data$  is the value of the  $i$ -th entry of  $a$ . (Once this point is made, there should be no objection to the  $a[i] = d$  notation, viewing it as just another way of writing the  $apply(a, i, d)$  formula.) When this approach is taken, it is necessary to state the properties of  $apply$ , for example that for every  $a$  in  $DataArray$  and  $i$  in  $UpdaterIndex$  there exists one and only one  $d$  in  $Data$  such that  $apply(a, i, d)$ .

2. The second approach can be called “set theoretic”. It requires to introduce the membership symbol  $\in$  and to transform our structures into models of (a small part of) set theory. When this is done, one can rely on standard definitions of pairs, functions etc. and give meaning to expressions such as  $a[i]$ . (See [8] for a more detailed description of set-theoretic system executions.)

We will not make a choice here and leave it to the reader to interpret  $a[i]$ , and to find a way to represent finite sequences.

We think of the set of indices  $UpdaterIndex$  as a finite set  $\{1, \dots, M\}$  of natural numbers, but this is not necessarily part of the specification. For every  $i \in UpraterIndex$  we denote with  $UpDater_i$  the  $i$ th updater process, and we identify this process with its set of events  $\{e \in Event \mid \mathbf{update}(e) \wedge index(e) = i\}$ . So,  $UpDater_i(e)$  can be thought of as a shorthand for  $\mathbf{update}(e) \wedge index(e) = i$ . The set of all events  $e$  such that  $\mathbf{update}(e)$  is called the  $UpDater$  process, and similarly the set of all events  $e$  such that  $\mathbf{scan}(e)$  is called the  $Scanner$  process. We shall use these two terms,  $UpDater$  and  $Scanner$  in an informal discourse, but there is no need to have them in the  $wss$  signature because they are expressible in that signature.

In the specification of [17] there are  $N$  serial processes, and each process  $i$  executes both scan and update operations. Here we prefer to separate the processes: there are scanner processors and updater processors. Each updater process is serial, but we do not assume here that  $Scanner$  is partitioned into serial processes and we view it as a single (non-serial) process. So the  $index$  function is actually a partial function from the events (it is defined only on the  $\mathbf{update}$  events). When we describe the protocol we will have serial scanner processes as in [17] (but we will still have distinct scanners and updaters). At this stage, however, when the notion of weak snapshot system is defined, we prefer to have a minimal set of assumptions that yield the definition.

Another function in the  $wss$  language is  $\Omega$ . Intuitively, for any scan operation  $S$  and updater index  $i$ ,  $\Omega(S, i)$  is that update operation by  $UpDater_i$

whose value  $S$  returned in its  $i$ th entry.

The specification of weak snapshot system will be given next (in Figure 1) in the *wss* language and called the list of *wss properties*. This is the explication of the [17] definition.

1. For every  $i$  in *UpdaterIndex*,  $UpDater_i$  is serial. That is, the set of events  $e$  such that  $\mathbf{update}(e) \wedge index(e) = i$  is linearly ordered by  $\rightarrow$ . No event falls under both **scan** and **update**.
2. For every **scan** event  $S$  and for every *UpdaterIndex*  $i$ , if  $U = \Omega(S, i)$  then  $\mathbf{update}(U)$ ,  $i = index(U)$ , and  $Value(S)[i] = Value(U)$ .
3.  $\Rightarrow$  is a partial (irreflexive) ordering on *Event* that extends the temporal ordering  $\rightarrow$ . That is,  $X \rightarrow Y$  implies that  $X \Rightarrow Y$ .
4. For every *UpdaterIndex*  $i$  and for every **scan** event  $S$ ,  $\Omega(S, i) \Rightarrow S$  and there is no  $UpDater_i$  event  $U$  with  $\Omega(S, i) \Rightarrow U \Rightarrow S$

Figure 1: List of *wss* properties.

### 3 Weak Snapshot Axioms

In section 7 we shall describe an algorithm that implements scan and update operations, and will prove that the resulting operations satisfy the *wss* properties. Instead of proving directly these properties, we will actually prove that any system execution of these operations satisfies some intermediary properties (called “*wss* axioms”) from which the *wss* properties follow. These axioms are used again in Section 10 where a self-stabilizing timestamp protocol is described which employs *Scan* and *UpDate* operations. The only assumptions needed for the protocol’s correctness is that these operations satisfy the *wss* axioms—their actual implementation is unspecified. So it seems that the *wss* axioms are of independent interest. The *wss* axioms are stated in the *wss* language in Figure 2.

**Theorem 3.1** *The wss intermediate axioms imply the wss properties in the following sense. There is a definable relation  $\Rightarrow$  so that if  $M$  is any system execution that satisfies the wss intermediate axioms, then the augmented*

1. For every  $i$  in  $UpdaterIndex$ ,  $Updater_i$  is serial. (Process *Scanner* is not necessarily serial.) The predicates **scan** and **update** are pairwise disjoint. (Recall that if  $U$  is an event, then we defined  $Updater_i(U)$  iff **update**( $U$ ) and  $i = index(U$ ).)
2. For every *Scan* event  $S$  and  $UpdaterIndex$   $i$ , if  $U = \Omega(S, i)$ , then:
  - (a)  $Updater_i(U)$  and  $Value(S)[i] = Value(U)$ .
  - (b)  $begin(U) < begin(S)$ , and  $end(U) < end(S)$ .
  - (c) If  $U'$  is any  $Updater_i$  event, then  $U \rightarrow U' \rightarrow S$  is impossible.

Figure 2: The *wss* intermediate axioms.

*model obtained by adding to  $M$  the definable relation  $\Rightarrow^M$  satisfies the wss properties.*

**Proof.** Properties 1 and 2 of the *wss* list of Figure 1 are stated explicitly in intermediate axioms 1 and 2(a). For every events  $X$  and  $Y$  define  $X \Rightarrow Y$  iff one or both of the following two possibilities occurs.

1.  $X \rightarrow Y$ ,
2. **scan**( $Y$ ) and  $X = \Omega(Y, k)$  for some  $UpdaterIndex$   $k$  (which implies that  $Updater_k(X)$ ),

It is obvious that  $\Rightarrow$  is irreflexive. Indeed  $X \rightarrow X$  is impossible since  $\rightarrow$  is irreflexive. But  $X = \Omega(X, k)$  is also impossible since  $X = \Omega(Y, k)$  implies that  $X$  is an *Updater* event, while  $Y$  is a *Scanner* event (and these two processes are disjoint).

**Lemma 3.2** *Relation  $\Rightarrow$  is transitive.*

To prove the lemma, just check all combinations for  $X \Rightarrow Y \Rightarrow Z$ , and conclude that  $X \Rightarrow Z$  for each one. There are four cases.

1.  $X \rightarrow Y$  and  $Y \rightarrow Z$ . Then  $X \rightarrow Z$  by transitivity of  $\rightarrow$  and hence  $X \Rightarrow Z$ . (Relation  $\rightarrow$  is transitive because we are in the context of system executions.)

2.  $X = \Omega(Y, k)$  and  $Y = \Omega(Z, \ell)$  is impossible since  $Y$  cannot be both a scan and an update.
3.  $X \rightarrow Y$  and  $Y = \Omega(Z, k)$ . Then  $\text{begin}(Y) < \text{begin}(Z)$  (by item b) and hence  $X \rightarrow Z$  follows.
4.  $X = \Omega(Y, k)$  and  $Y \rightarrow Z$ . Then  $X \rightarrow Z$  follows from  $\text{end}(X) < \text{end}(Y)$ .

■ The main point (in proving Theorem 3.1) is to prove that there is no event  $U$  in  $\text{UpDater}_i$  such that  $\Omega(S, i) \Rightarrow U \Rightarrow S$ . Suppose on the contrary that there is one. The only possibility for  $\Omega(S, i) \Rightarrow U$  is that  $\Omega(S, i) \rightarrow U$  (for the remaining possibility implies that  $U$  is a *Scan*). What are the possibilities for  $U \Rightarrow S$ ?

1.  $U \rightarrow S$  is impossible, since  $\Omega(S, i) \rightarrow U \rightarrow S$  contradicts intermediate axiom 2(c).
2.  $U = \Omega(S, i)$  is clearly impossible since  $\Omega(S, i) \rightarrow U$ , and  $\rightarrow$  is irreflexive.

■

## 4 Executions of algorithms

We have seen that Tarskian system executions, as defined in 2.2 can be used to explicate higher-level properties of systems, and to formally derive properties from some other properties (such as the *wss* list from the *wss* intermediate axioms). System executions can also be used to specify communication devices (for example registers), to define executions of concurrent algorithms, and to prove that these executions satisfy certain properties. This will be needed here for we shall describe a *Scan/UpDate* weak snapshot algorithm in Section 7, and prove that all its system executions stabilize and finally satisfy the *wss* intermediate axioms. A complete and detailed definition of what consists an execution of a concurrent algorithm would be out of place in this article, and we shall only describe some salient points. (The interested reader will find in [4] a general definition that suits our paradigm.)

We begin by specifying registers. There are atomic, regular, and safe registers in the algorithm described in this paper. For simplicity, we assume

that the atomic registers are, in fact, serial. That is, that the read/write events on any atomic register are linearly ordered and every read returns the value of the last preceding write.

All our registers here are single writer multiple reader registers. That is, register  $R$  has a single “owner” which is a serial process that executes write actions on  $R$ , and is hence called the Writer of  $R$ , and there are several reading processes, collectively called Reader.

We find it convenient to assume a “return” function  $\omega$  (not to be confused with the set of natural numbers, also denoted  $\omega$ ). This is useful mainly for regular registers, but will also be employed for serial and safe registers. For each read event  $x$  of a register  $R$ ,  $\omega(x)$  is a write event on that register, and the following requirements define the regularity of  $R$ :

1. For every read  $x$  we have  $Value(\omega(x)) = Value(x)$ . That is, the value returned by the read  $x$  is the value of that write that has had its impact on  $x$ .
2. For any read event  $x$  it is not the case that  $x \rightarrow \omega(x)$ .
3. There is no write event  $w$  on register  $R$  such that  $\omega(x) \rightarrow w \rightarrow x$ .

In order to formally define the behavior of a regular register,  $R$ , we have to define first an appropriate system execution signature  $L_R$ . This signature contains two unary predicates: *read* and *write* defined over the sort *Event*; a function *Value* from *Event* to an atemporal sort called *Data*, and a function  $\omega$  defined on *Event*. In addition, the signature contains those items that are always present in a system execution signature (Definition 2.1). Actually, in presence of several registers, we may need an additional predicate  $R$  (the name of the register) to express the fact  $R(e)$  that event  $e$  is a read/write event on register  $R$ . Then a system execution  $M$  that interprets this signature models a regular register if it satisfies the properties described in items 1-3 above (which should formally be written in the language  $L_R$  rather than in mathematical English).

The notion of “regularity” is quite useful. We say that a function  $f : A \rightarrow B$  is *regular* (where  $A$  and  $B$  are sets of events) iff for every  $a \in A$  it is not the case that  $a \rightarrow f(a)$ , and there is no event  $b \in B$  such that  $f(a) \rightarrow b \rightarrow a$ .

Safe registers are defined by the demands that the equality  $Value(\omega(x)) = Value(x)$  holds whenever read  $x$  is not concurrent with any write. (Events  $a$  and  $b$  are not concurrent if  $a \rightarrow b$  or  $b \rightarrow a$ .)

We continue with some words on the semantics of programs that employ communication devices. We deal in this paper with protocols that implement certain communication operations of a certain type (let's call it  $\beta$ ) using communication devices of another type called  $\alpha$ . For example, the protocols of section 5 implement *Scan* and *UpDate* operations using atomic, regular, and safe registers. So the  $\alpha$  actions are the assumed register read and write operations, and the  $\beta$  operations are the implemented *Scan/UpDate* operations. These  $\beta$  operations become the  $\alpha$  actions in a later section (10) in which a self-stabilizing time stamp protocol is described. That protocol implements *SCAN/LABEL* operations with operation-algorithms that use *Scan/UpDate* actions (and regular registers). So in Section 10 the  $\beta$  operations are the *SCAN/LABEL* operations, and the  $\alpha$  operations are the *Scan/UpDate* and regular read/write operations. When we prove the correctness of such protocol we have operation-algorithms that implement those  $\beta$  operations using some  $\alpha$  type devices. We assume several serial processes  $P_1, \dots, P_k$ , and each process executes a finite or infinite number of times its  $\beta$  operation-algorithm, each execution with some parameter (For example, the index of the executing process is one of the parameters, and for a "writing" operation the new value is another.) A single execution of an operation-algorithm, that is a  $\beta$  operation execution, results in a higher-level event comprised of lower-level  $\alpha$  events. We must prove that the resulting  $\beta$  high level events satisfy their specification, and we can use in the proof the assumption that the assumed  $\alpha$  actions satisfy their specifications.

Following [4], we base our modeling approach on the distinction between restricted and unrestricted semantics. By *unrestricted semantics* we mean analysis of the *forms* of programs which relates these forms to meaning, without considering the external world and the communication devices. For example, it is part of unrestricted semantics to define that the execution of a sequence of two instructions  $p_1; p_2$  is done by first executing  $p_1$  and then (if it terminates) executing  $p_2$  in such a way that its starting state is the state in which  $p_1$  has terminated. The notion of state is of prime importance here, but only local states of each process matter for the unrestricted semantics. The term "unrestricted semantics" is used because the communication devices are not restricted to operate properly: a read of a register, for example, may obtain any value (within its range) even if that value has never been written. As far as the specification of each process algorithm is concerned (detached from the activities of other processes) the proper functioning of the communication devices is immaterial.

By *restricted (or external) semantics* we mean the specification of communication devices and the relationship of the programs with the external environment. The semantics of registers, queues, or channels, for example, do not depend on the programs that use them, and ought to be described separately from the programs' semantics. In describing the semantics of communication devices and how it affects the programs, system executions are of prime importance. The term "restricted semantics" is used to convey the idea that the communication devices operate properly.

For a single process  $P$  (always a serial process) the unrestricted semantics of its algorithm is simple. The easiest description is by means of local states, transitions, and histories.

1. There is a list of local variables of  $P$ , and a local state is a map which assigns to each of these variables a value. One of the variables is a program-counter which points to the instruction to be executed. The state of the process describes its local variables but not the states of the communication devices. (The registers for example are not part of its state).
2. To each execution of an instruction in the operation-algorithm there corresponds a set of possible transitions. A transition is a pair of states: the states before and after the instruction action. For example, when  $R$  is a register and  $x$  a variable, then an instruction "read  $x := R$ " corresponds to all pairs of states  $\langle S, T \rangle$  where states  $S$  and  $T$  agree on all variables, except possibly at  $x$ . The new value  $T(x)$  is the value obtained in this execution of the read of register  $R$ . Since  $R$  itself is not a local variable, the return value is arbitrarily chosen and  $T(x)$  can be any value in the range of values of  $R$ .
3. An unrestricted history of a process  $P$  is a sequence (finite or infinite) of local states of  $P$ , beginning with some initial state, such that each pair of a state and its successor in this sequence is a transition of the operation-algorithm executed by  $P$ .

A collection of  $k$  unrestricted histories, one for each process  $P_i$ , represents a possible unrestricted execution of the system. The fact that a read of a regular register, for example, in a local unrestricted history can return any value, not even corresponding to a write on that register, is a characteristic of unrestricted executions.

To obtain restricted executions, those in which the  $\alpha$  communication devices operate properly, we must rely on system executions since it is by means of these structures that the  $\alpha$  devices were specified, and since the correctness conditions for the  $\beta$  operations are stated by reference to system executions. (See for example the regular-register specifications given above, and the *wss* properties of Figure 2.) Histories are basically sequences of states, while system executions are Tarskian structures, and hence the histories used for the unrestricted semantics would be incongruous with the system executions needed for the device specifications. Since a uniform platform is needed to support the correctness proof at all its levels, the history sequences that represent the unrestricted semantics are defined as system executions in [4], and then the unrestricted semantics of an algorithm is a collection of system executions that describe executions of the algorithm in which there is no restriction on the communication devices used. The restricted semantics is a subset of that collection in which each communication device operates properly. Now a system execution that satisfies both the unrestricted semantics (of the algorithm) and the restricted semantics (of the  $\alpha$  communication devices) is an execution of the algorithm, and the mission of the correctness proof is to show for every such system execution that the implemented  $\beta$  operations satisfy their specifications.

## 4.1 High-level events

As explained, all processors execute their protocol (or operation-algorithm, these notions are synonymous here) any number of times. The execution of an operation-algorithm contains accesses to shared communication devices. That is,  $\alpha$  events corresponding to the instructions of the algorithm. So there are two kinds of events: low level events ( $\alpha$  actions) and higher level events which are the operation executions, that is the  $\beta$  executions. Now we have the following difficulty when we come to prove the correctness of an operation protocol: the correctness conditions are expressed in term of the resulting high level events (for example the *wss* properties) but these properties certainly depend on the low level events, namely on the execution of the protocol. To enable a smooth proof, it seems reasonable to represent both the low and the high level events in the same structure. This discussion leads to the following definition.

**Definition 4.1 (System execution with high level events)** *Let  $M$  be a*

system execution. We say that  $M$  is a system execution with high level events if the signature of  $M$  contains a binary “membership” symbol  $\in$  defined on the events of  $M$  with the following properties.

1. For every event  $a$  in  $M$  there are two possibilities: either there is no  $x$  such that  $x \in a$  holds in  $M$ , and then  $a$  is said to be a low level event, or else there is some  $x$  such that  $x \in a$  holds, and then  $a$  is said to be a high level event. In this case, if  $x \in A$  holds, then  $x$  is necessarily a low level event.<sup>2</sup>
2. Extensionality holds for the high level events: if  $A$  and  $B$  are high level events such that, for every  $x$ ,  $x \in A$  iff  $x \in B$  then  $A = B$ . (Low level events always have the same members—that is no members—so extensionality does not hold for low level events.)
3. If  $A$  is a high level event then its members are linearly ordered in the precedence ordering relation  $\rightarrow$ , and  $A$  contains only a finite number of low level events (equivalently  $A$  has a last low level event). The functions  $begin$  and  $end$  are defined on high level events as well. If  $A$  is a high level event and  $a \in A$  is its first lower level event then  $begin(A) = begin(a)$ , and if  $b \in A$  is its last event then  $end(A) = end(b)$ . It follows that for every high level events  $A$  and  $B$ ,  $A \rightarrow B$  iff  $\forall x \in A \forall y \in B (x \rightarrow y)$ .

In all applications brought in this paper, every low level event belongs to at most one higher level events. So if  $a$  is any low level event we denote with  $[a]$  that high level event such that  $a \in [a]$ . Also, in our applications, if  $A$  is any high level event then all its members belong to the same processor and we have that the index of  $A$  ( $index(A)$ ) is the same as that of any of its member.

## 5 A weak snapshot with unbounded values

We bring in this section a variant of the unbounded weak snapshot algorithm described in [17]. The unbounded algorithm is much simpler than the bounded one, and the issue of self-stabilization does not appear. This

---

<sup>2</sup>It is conceivable that a set of events contains an event that is itself a set of events etc., but in this paper there is no need for such events and two “layers” suffice. Hence our restriction.

simplicity is useful because it allows us to illustrate some of the concepts described here (such as the weak snapshot intermediate axioms and system executions) in a simpler setting and with less formal details.

The unbounded weak snapshot algorithms for the update and the scan operations are shown in figures 3 and 4.

There are  $M$  *Updater* processes:  $Updater_i$  for  $i = 1, \dots, M$ . There are  $N$  *Scanner* processes:  $Scanner_j$  for  $j = 1, \dots, N$ . Data types used by the protocol are as follows:

1. *Data* is the type of memory locations, that is, the values that are written by the *Updater* and read by the *Scanner*.
2. *Color* is the set of integers.
3. *Head* is a record type. If  $h$  is in *Head* then it has the following fields.
  - (a)  $h.data$  is a *Data* value.
  - (b)  $h.colors$  is an array of length  $N$ ; each  $h.colors[j]$  (for  $1 \leq j \leq N$ ) is in *Color*.

The registers used are the following.

1. Each  $Scanner_j$ , for  $1 \leq j \leq N$ , writes on a single-writer multiple reader atomic register  $ScUp_j$  (read by each *Updater*). The values of  $ScUp_j$  are natural numbers (that is colors).
2. Process  $Updater_i$ , for  $1 \leq i \leq M$ , writes on the following registers.
  - (a) A multiple reader atomic register  $H_i$  of type *Head* that is read by all *Scanners*.
  - (b)  $Updater_i$  has  $N$  safe registers (also called buffers) of type *Data*.  $B_{i,j}$  is written by  $Updater_i$  and is read by  $Scanner_j$ .

Atomicity of a register means serializability of its read/write events. For simplicity we assume that registers  $H_i$  and  $ScUp_j$  are already serial in all the executions considered.

This unbounded algorithm is essentially the one brought [17] and the main reason it is reproduced here is to serve as an introduction to the more complex stable and bounded algorithm described in this paper. Some features and properties are shared by both algorithms, and their understanding is easier

*UpDate*(*data*: *Data*) (\* by *UpDater<sub>i</sub>* \*)

1. *old\_colors* := *colors*;
2. forall  $1 \leq j \leq N$ 
  - (a) read *colors*[*j*] := *ScUp<sub>j</sub>*;
  - (b) if *colors*[*j*]  $\neq$  *old\_colors*[*j*] then write *B<sub>i,j</sub>* := *old\_data*;
3. *old\_data* := *data*;
4. write *H<sub>i</sub>* :=  $\langle$ *data*, *colors* $\rangle$ .

Figure 3: The unbounded *UpDate* algorithm for *UpDater<sub>i</sub>* ( $1 \leq i \leq M$ ). Local variables are: *colors*, *old\_colors* are arrays of length *N* of colors. *data* and *old\_data* are in *Data*. The initial value of variable *old\_data* is *d<sub>0</sub>*, which is the initial value of the buffers.

*Scan* (\* by *Scanner<sub>j</sub>* \*)

1. write *ScUp<sub>j</sub>* := *color*;
2. forall  $1 \leq i \leq M$ 
  - (a) read *h*[*i*] := *H<sub>i</sub>*;
  - (b) if *h*[*i*].*colors*[*j*] = *color* then read *values*[*i*] := *B<sub>i,j</sub>*  
else *values*[*i*] := *h*[*i*].*data*;
3. *color* := *color* + 1;
4. return *values*.

Figure 4: The *Scan* algorithm for *Scanner<sub>j</sub>*. Local variables are: *color* a natural number. *values* is an array of length *M* of *Data* values; *h* is an array of length *M* of *Head* values.

in the simpler context of the unbounded algorithm. In particular, executions of the unbounded algorithm satisfy the intermediary axioms of Section 3, and by Theorem 3.1 the unbounded algorithm satisfies the weak snapshot properties.

Let  $M$  be a system execution that describes a run of the unbounded Scan/Update algorithm. We want to prove that the resulting high level events (*Scan* and *Update* operation executions) satisfy the intermediate axioms and the first step in that proof is to define the function  $\Omega$

## 6 Stabilization of system executions

Usually, self-stabilization of a system is defined in terms of its global states. Some states are defined to be “good” and self-stabilization means that if the system starts in an arbitrary state then the system will eventually reach a good state. By “arbitrary state” one means a state in which some of the processes are crashed, and then they will never recover, and some other processes are active (non-crashed) and act normally as dictated by their programs (and they will never crash). An active process needs not start from the first instruction of its protocol, and the initial values of its local and global variables are arbitrarily determined in this initial state (yet all variables are in their types).

This definition is not suitable for our approach because we concentrate on behavior rather than state. While a behavior is extended in time, a state is the description of an instant (formally it is a map that gives values to all variables of the system). In behavioral terms, self-stabilization of a system means that any erratic behavior will eventually turn into an ordinate one. A behavior is for us a property of system executions, usually system executions that represent higher-level events. It is the collection of all system executions that interpret some specific language signature and satisfy a certain sentence in that language, a sentence which intuitively expresses a desirable good property of the execution. For example, the collection of all system executions that satisfy the *wss* properties is such a behavior specification. (There is not much difference between a property and its extension, between a property of system executions and the collection of all system executions that satisfy this property.)

It is possible that a system execution does not satisfy the desirable behavior, but it does so after a certain moment  $m$ . We say then that this

particular system execution stabilizes to the desirable behavior. For example, a temperature gauge may display erratic behavior for some time, and then stabilizes and reports accurately the temperature of the object it is supposed to monitor.

Let  $L$  be some system execution language, and  $P(x)$  be a formula in  $L$  with a free variable  $x$  of sort *Moment*.

**Definition 6.1** *We say that a system execution  $M$  (an interpretation of  $L$ ) stabilizes for  $P$  (or that  $P$  stabilizes in  $M$ ) iff for some moment  $m$   $P(m)$  holds in  $M$ . Intuitively,  $P(m)$  says that some good behavior holds after  $m$ .*

We think of  $P(x)$  not so much as a statement about  $x$  but rather as one about all events that begin after  $x$ . Most properties that are investigated for selfstabilization are such that if  $P(m_0)$  holds for any  $m_0$  then  $P(m')$  holds for every  $m' \geq m_0$ , and hence it suffices to prove the existence of one moment  $m_0$  such that  $P(m_0)$  in order to derive stabilization.

Our aim is to define when an algorithm self-stabilizes to property  $P(x)$ . We can try to say that stabilization to  $P$  occurs if in every system execution of that algorithm there exists a moment  $m$  such that  $P(m)$  holds. But this would not be true, because there are some obvious assumptions that have to be made in order for the algorithm to reach its stabilization: for example, that the communication devices employed by the algorithm stabilize. Suppose for example an algorithm that uses regular registers (such as the timestamp algorithm described in the second part of this paper). If the registers continue to malfunction then no algorithm can stabilize and overcome this nonterminating malfunctioning. We must assume that at some moment the registers (and any other communication devices) behave normally.

So, for every communication device  $\text{Com}$  we will define a property  $P_{\text{Com}}(x)$  which says that the device behaves normally after moment  $x$ . We say that  $\text{Com}$  “stabilizes” in  $M$ , a system execution, if  $P_{\text{Com}}$  stabilizes, that is, by the definition above, for some moment  $m$   $P_{\text{Com}}(m)$  holds in  $M$ . Then for any algorithm  $A$  that uses communication devices  $\text{Com}_i$ , for  $i = 1, \dots, k$ , we will say that  $M$  is a “normal” system execution of  $A$  if every  $\text{Com}_i$  stabilizes in  $M$ , and there exists a moment  $m$  so that every non-crashed process executes its protocol in an orderly way after  $m$ . The statement that “Algorithm  $A$  self-stabilizes to property  $P$ ” will then mean that in any normal system execution  $M$  of algorithm  $A$ , there exists a moment  $m$  such that  $P(m)$  holds.

There are still some fine points in these definitions that have to be clarified, and we will do that in the following subsections. We begin by specifying

$P_{\text{Com}}$  for the different communication devices employed by the algorithms in this paper.

We shall describe two self-stabilizing algorithms in this paper: a weak snapshot algorithm (in Section 7) and a timestamp algorithm (in Section 10). The first protocol employs serial and regular register (supporting read/write actions), and the second protocol employs regular registers and weak snapshot registers (which support *Scan* and *UpDate* actions).

## 6.1 Stable regular registers

We define here the stabilization property of a regular register in a system execution. Regular registers were defined in 4. There is a serial *Writer* process who owns register  $R$ , and a *Reader* process (comprising several subprocesses, so that *Reader* is not serial). The requirements for regularity were expressed by means of a return function  $\omega$ . For every *read* event  $x$ :

1. It is not the case that  $x \rightarrow \omega(x)$ .
2. There is no *write* event  $w$  on register  $R$  such that  $\omega(x) \rightarrow w \rightarrow x$ .
3. For every *read*  $x$  we have  $\text{Value}(\omega(x)) = \text{Value}(x)$ . That is, the value returned by the read  $x$  is the value of that write that has had its impact on  $x$ .

For self-stabilization we have to consider the possibility that *Writer* is crashed and contains a finite number of *write* events. In this case, we want for stabilization to have a moment  $m$  so that all reads that begin after  $m$  return the same value. In other words, the crashed *Writer* has to leave its register in a determined (yet arbitrary) value which all reads finally return. The most convenient way to express this is by assuming an “ultimate” write event whose value all reads finally obtain.

Again for self-stabilization we have to consider the possibility that a bounded initial segment of the execution is very inordinate, including the possibility that *Writer* is not serial in that segment and contains concurrent writes. If the writer is not crashed, then it must finally become a serial process of course.

For any moment  $m$  let  $F_m$  denote the set of all *read/write* events that begin after  $m$ . Now  $P_R(m)$  is the following formula. (We use the notation  $A \rightarrow = B$  which means “ $A \rightarrow B$  or  $A = B$ .”)

1. The function  $\omega$  is defined on all the *read* events  $r$  in  $F_m$ ,  $\omega(r)$  is a *write* event (in *Writer*) in this case and  $Value(r) = Value(\omega(r))$  and it is not the case that  $r \rightarrow \omega(r)$ .

Define  $G_m = F_m \cup \{e \mid \exists r \in F_m (read(r) \text{ and } \omega(r) \rightarrow e)\}$ . Then the *Writer* events in  $G_m$  are serially ordered.

2. If  $r$  is a *read* event in  $F_m$  then there is no *Writer* event  $w$  such that  $\omega(r) \rightarrow w \rightarrow r$ .
3. If *Writer* is crashed (contains a finite number of events), then all its events precede  $m$ .

Notice that the set  $G_m$  is a final segment of events, in the sense that if  $e_1 \rightarrow e_2$  and  $e_1 \in G_m$  then  $e_2 \in G_m$  as well.  $G_m$  contains all the events that start after  $m$  as well as those that are needed to explain the values returned by the reads. That is, write events  $\omega(r)$  for  $r \in F_m$  are introduced into  $G_m$  (and in addition, if  $e$  follows  $\omega(r)$  then  $e$  is also introduced into  $G_m$  so as to make  $G_m$  a final segment).

We explain this specification  $P_R(m)$  of register  $R$ . Consider any read event that begin after moment  $m$ , namely a read in  $F_m$ . It is conceivable that  $w_0 = \omega(r)$ , the corresponding write, is not in  $F_m$ , either because  $w_0$  precedes  $m$  or because  $begin(w_0) \leq m \leq end(w_0)$ . In any case,  $\omega(r)$  is in  $G_m$  as well as any write that follows it. In case *Writer* is crashed we get that  $\omega(r_1) = \omega(r_2)$  for every read events  $r_1$  and  $r_2$  in  $F_m$ . (Because  $\omega(r_1), \omega(r_2) \in G_m$ , and if unequal then  $\omega(r_1) \rightarrow \omega(r_2)$  or  $\omega(r_2) \rightarrow \omega(r_1)$  by seriality (item 1). But both  $\omega(r_1)$  and  $\omega(r_2)$  end before  $m$  (item 3) and a contradiction to regularity (item 2) is obtained.) This *Writer* event,  $\omega(r)$  where  $r \in F_m$  is any read, is called the ultimate write of the crashed *Writer*. Of course, it is possible that *Writer* is not crashed.

A specification of a communication device such as a register can be seen as a guarantee made by the manufacturer of the device to the user saying that the register will behave in accordance with that specification. When we prove the correctness of an algorithm that employs certain communication devices, we are justified in assuming that the communication devices behave correctly and we assume that the specifications hold in the system execution that we analyze. So, when we prove that an algorithm that uses regular registers is self-stabilizing, we assume that there exists a moment  $m$  so that every regular register  $R$  satisfies  $P_R(m)$ , and use this assumption in the proof.

There is a simple point that has to be mentioned nonetheless. If, by mistake, the algorithm contains two concurrent processes  $P_i$  and  $P_j$  that repeatedly write on the same register  $R$ , and as a result the set of write events is not serially ordered and contains an infinite pairs of concurrent writes, then we have no right to complain to the manufacturer of the register. We misused the register and the blame is on us. Therefore, in a correctness proof, it is the obligation of the prover to check that each register has a single writing process—a trivial check of course. Only then can we expect that the registers self-stabilize and that  $P_R(m)$  holds for some moment  $m$ .

## 6.2 Stable weak snapshots

Weak snapshot **update** and **scan** operations were specified in Figure 1 (the list of *wss* properties). Here we define what it means that these properties stabilize in a system execution. That is we define property  $P_{wss}(m)$  which says, intuitively, that “the *wss* properties hold after  $m$ ”. As a first approximation we may try to take the universe of all events that begin after  $m$  and to apply the *wss* properties with a restriction of the quantifications to that universe. But then, it is possible that  $S$  is a **scan** event that begins after  $m$  and yet, for some index  $i$ ,  $\Omega(S, i)$  does not begin after  $m$ . Clearly we want to add  $\Omega(S, i)$  in this case to the universe of discussion because it is needed there to explain the value returned by  $S$ . In case  $UpDater_i$  is crashed,  $\Omega(S, i)$  is the “ultimate” event of  $UpDater_i$  and we add it to the universe of our stabilized structure, but when  $UpDater_i$  is not crashed, it is necessary to add not only  $\Omega(S, i)$  but also any **update** event that follows it. For any moment  $m$ , let  $F_m$  be the set of **scan** and **update** events  $x$  that begin after  $m$  (that is,  $m < begin(x)$ ) and define  $G_m$  to be the union of  $F_m$  with the set of all **update** events  $U$  such that, either  $UpDater_i$  is crashed and  $U$  is the ultimate  $UpDater_i$  event, or else  $UpDater_i$  is non-crashed and  $\Omega(S, i) \rightarrow= U$  for some **scan** event  $S$  in  $F_m$ .

$P_{wss}(m)$  is the conjunction of the following statements.

- 1 <sub>$m$</sub>  For every  $i$  in  $UpdaterIndex$ ,  $UpDater_i$ , restricted to  $G_m$ , is serial. No event falls under both **scan** and **update**.
- 2 <sub>$m$</sub>  For every **scan** event  $S$  in  $F_m$ , and for every  $UpdaterIndex$   $i$ , if  $U = \Omega(S, i)$  then **update**( $U$ ),  $i = index(U)$ , and  $Value(S)[i] = Value(U)$ .

- 3<sub>m</sub> Relation  $\Rightarrow$  is a partial (irreflexive) ordering on  $G_m$  that extends the restriction of the temporal ordering  $\rightarrow$  to  $G_m$ .
- 4<sub>m</sub> For every *UpdaterIndex*  $i$  and for every **scan** event  $S$  in  $F_m$ ,  $\Omega(S, i) \Rightarrow S$  and there is no *Updater<sub>i</sub>* event  $U$  with  $\Omega(S, i) \Rightarrow U \Rightarrow S$ .
- 5<sub>m</sub> If *Updater<sub>i</sub>* is crashed (contains a finite number of events) then all its events precede  $m$ .

The conjunction of properties 1<sub>m</sub>, 2<sub>m</sub>, 3<sub>m</sub>, 4<sub>m</sub>, 5<sub>m</sub> is defined to be the statement that “the *wss* properties hold after  $m$ ”. It follows in case *Updater<sub>i</sub>* is crashed that for every **scan** events  $S_1$  and  $S_2$  in  $F_m$  that  $\Omega(S_1, i) = \Omega(S_2, i)$  and this common value is called the ultimate *Updater<sub>i</sub>* event.

### 6.3 Stable *wss* intermediate axioms

The weak snapshot intermediate axioms were defined in Figure 2 in Section 3, and we proved in Theorem 3.1 that they imply the *wss* properties. In this subsection we define property  $P_{wss-axioms}(m)$  which says intuitively that the intermediate axioms hold after moment  $m$ . For any moment  $m$ , let  $F_m$  be the set of all events that begin after  $m$ , and define,  $G_m$  as in (6.2). Then  $P_{wss-axioms}(m)$  is the conjunction of the following statements.

- 1<sub>m</sub> For every  $i$  in *UpdaterIndex*, the events of *Updater<sub>i</sub>* that are in  $G_m$  are serially ordered. The predicates **scan** and **update** are pairwise disjoint.
- 2<sub>m</sub> For every *Scan* event  $S$  that begins after  $m$  and *UpdaterIndex*  $i$ , if  $U = \Omega(S, i)$ , then:
  - (a)  $Updater_i(U)$  and  $Value(S)[i] = Value(U)$ .
  - (b)  $begin(U) < begin(S)$ , and  $end(U) < end(S)$ .
  - (c) If  $U'$  is any *Updater<sub>i</sub>* event, then  $U \rightarrow U' \rightarrow S$  is impossible.
- 3<sub>m</sub> Any crashed process contains only events that precede  $m$ .

Figure 5: The  $P_{wss-axioms}(m)$  statement, saying essentially that the *wss* intermediate axioms hold after moment  $m$ .

Theorem 3.1 can be extended (with almost no change in the proof) to show that  $P_{wss-axioms}(m)$  implies that  $P_{wss}(m)$ . That is, stabilization of the

<p>Algorithm for process <math>P_i</math>:</p> <p>repeat forever</p> <ol style="list-style-type: none"> <li>1. compute the parameters needed;</li> <li>2. invoke the protocol for operation <math>OP_i</math> with the parameters computed;</li> </ol>
--

Figure 6: The generic algorithm for  $P_i$

intermediate axioms bring about stabilization of the weak snapshots. The stabilizing intermediate axioms have an important place here. On one hand, we will prove that our *wss* protocol in Section 7 stabilizes for  $P_{wss-axioms}$ , and on the other hand the timestamping algorithm of Section 10 will be shown to work correctly under the assumption that its **scan/update** actions stabilize for  $P_{wss-axioms}$ . So that these intermediate axioms appear here both as an aim and means.

## 6.4 Normal executions and stabilization

The two algorithms presented in this paper have the following form. There are serial processes  $P_1, \dots, P_k$ , and correspondingly a set of operations  $OP_1, \dots, OP_k$ , where each operation  $OP_i$  is implemented by some operation-algorithm which is called the protocol for  $OP_i$ . Each process  $P_i$  executes the algorithm described in Figure 6

In this generic algorithm process  $P_i$  invokes operation  $OP_i$ , and we assume that the protocol for operation  $OP_i$  is always terminating, unless  $P_i$  crashes. Thus, a non-crashed process  $P_i$  invokes the protocol for  $OP_i$  an infinite number of times. The aim of the protocols for the operations  $OP_i$  is to ensure that some good property  $P$  holds, and, for stabilization, the aim is to ensure that property  $P$  finally holds in any normal system execution of the algorithm. For a formal definition we assume that the processes employ a certain set of communication devices (the  $\alpha$  devices), and formula  $P_\alpha(m)$  expresses the assumption that all communication devices in this set work properly after moment  $m$ . We have another formula  $P(m)$  which expresses the fact that the desirable property of the system holds from moment  $m$  on. In this situation, the stabilization of the algorithm for property  $P$  essen-

tially says that  $\exists m P_\alpha(m) \rightarrow \exists m P(m)$ . Namely,  $P(m)$  finally holds for some moment  $m$ , provided that the assumed communication devices finally work correctly. For a more detailed definition of stabilization, we first need the following.

**Definition 6.2** *Let  $M$  be a system execution with events that are partitioned into “processes”  $P_1, \dots, P_k$ . Some processes are finite (contain a finite number of events) and some are infinite. A finite process is said to be crashed and an infinite process is said to be non-crashed. We say that  $M$  is “normal” if the following holds for some moment  $m$ .*

1.  $P_\alpha(m)$  holds. That is, all communication devices in  $M$  behave correctly according to their specifications after  $m$ .
2. Every crashed process contains only events that have ended before  $m$ .
3. After moment  $m$ , the events of any non-crashed process  $P_i$  represent executions of operations  $OP_i$  that are done in accordance with its operation-algorithm.
4. The high-level events of  $M$  consist of the following events. For a non-crashed  $P_i$ , the events are those high-level events that represent justified executions of a  $OP_i$  operation, in accordance with its protocol (a justified execution is one that begins with the first instruction of the protocol). And for a crashed  $P_i$  there is a single high-level event, called the ultimate event of  $P_i$ . It consists of all ultimate actions on the  $\alpha$  devices of the crashed process. (In our case, the ultimate event of a crashed  $P_i$  consists of all write and **update** events on the registers of  $P_i$ ).

**Definition 6.3** *We say that an algorithm is self-stabilizing with respect to some property  $P(x)$  when every normal system execution of the algorithm contains a moment  $m_1$  so that  $P(m_1)$  holds.*

Concerning Definition 6.2 and our identification of crashed processes with those that have a finite number of events, one may argue that a process can stop and have a finite number of events without necessarily being crashed or malfunctioning. That is certainly a valid argument, but for self-stabilization our distinction makes sense.

We want to clarify item 4 in Definition 6.2. Usually the desirable property  $P(x)$  refers to the higher-level operation executions, and the question

is how are these events defined? When self-stabilization is not the issue, the definition of the higher-level events (operation executions) is obvious. Each invocation of the protocol of  $OP_i$  generates a finite number of events (the actions corresponding to the instructions of the protocol's text), and the collection of these actions (lower-level events) forms a higher-level event that represent that invocation. For self-stabilization, however, we assume a transient period of totally inordinate behavior of the system in which it is impossible to assemble the events into a coherent higher-level event. (For example, the process may mistakenly execute a different algorithm, or an arbitrary sequence of actions.) So we must agree that the definition of the higher-level events of a normal system execution include only those sets of lower-level events that represent complete justified executions of the  $OP_i$  protocol (that start from the first instruction of that protocol) with the addition of an ultimate high-level event for each crashed process. That ultimate event of processor  $P_i$  is the set of all write and **update** ultimate events on the registers of  $P_i$ .

Let  $M$  be a normal system execution. We know (by Definition 6.2(1)) that there is a moment  $m_0$  so that communication works correctly after  $m_0$ , all events by a crashed process have ended before  $m_0$ , and the events of each process that start after  $m_0$  describe a correct execution of the algorithm of that process. Still, it is possible that the local variables of that process are somehow not "in tune" because of previous erratic behavior of the process. For example, the *UpDate* protocol of Figure 8 contains a local variable *parity*, and the value of that variable in any execution of *UpDate* is determined as the value set by the previous execution (or an initial value for the first execution). As a result, process  $P_i$  writes on buffers  $B_i^0$  and  $B_i^1$  alternately. But if we admit an inordinate beginning, then it is possible to have two operation executions  $E_1$  followed by  $E_2$  that both write on the same buffer, even though  $E_2$  begins after stabilization moment  $m_0$ . The point being that  $E_1$  has determined the wrong value of *parity*. For this reason we need a later moment which gives us greater assurance. There is a moment  $m'_0 > m_0$  so that for every non-crashed process  $p$  there is an operation execution (high-level event)  $X$  by  $p$  so that

$$m_0 < \text{begin}(X), \text{ and } \text{end}(X) < m'_0. \quad (2)$$

The usefulness of moment  $m'_0$  is to ensure that if operation execution  $Y$  by  $p$  has not ended before  $m'_0$ , then  $Y$  has a predecessor operation execution  $X$  by  $p$  and that predecessor started after  $m_0$ .

Considerations of this type may lead to some later moment  $m_{stable} > m'_0$  by which we may be assured that the system stabilizes.

**Definition 6.4** *If  $M$  is a normal system execution, then any moment  $m$  in  $M$  such that  $m > m_{stable}$  is said to be a normal moment.*

In the sequel we will define a *Scan/UpDate* algorithm and prove that if  $m$  is a normal moment in a normal system execution, then the *wss* intermediate axioms hold after  $m$ . Namely we will prove that  $P_{wss-axioms}(m)$  holds. This implies that the *wss* properties hold after  $m$ , and therefore that the system stabilizes for the *wss* properties.

## 7 A self-stabilizing weak snapshot algorithm

Our self-stabilizing weak snapshot algorithms for the *UpDate* and the *Scan* operations are shown in figures 8 and 9.

There are  $M$  *UpDater* processes:  $UpDater_i$  for  $i = 1, \dots, M$ . There are  $N$  *Scanner* processes:  $Scanner_j$  for  $j = 1, \dots, N$ . Data types used by the protocol are as follows:

1. *Data* is the type of memory locations, that is, the set of values that are written by the *UpDater* and read by the *Scanner* processes.
2. *Color* is the set of integers  $\{1, \dots, 5\}$ . We will see later why five colors are necessary.
3. *UpSc* is a record with two fields of type *Color*. If  $u$  is in *UpSc*, then  $u.old\_color$  and  $u.color$  are its two fields. When we write  $v = \langle a, b \rangle$  for an *UpSc* value  $v$  we mean that  $v.old\_color = a$  and  $v.color = b$ .
4. *Head* is a record type. If  $h$  is in *Head* then it has the following fields.
  - (a)  $h.colors$  is an array of length  $N$ ; each  $h.colors[j]$  (for  $1 \leq j \leq N$ ) is in *Color*.
  - (b)  $h.equalities$  is an array of length  $N$ ; each  $h.equalities[j]$  (for  $1 \leq j \leq N$ ) is boolean (*true* or *false*).
  - (c)  $h.parity$  is 0 or 1.

The registers used are the following.

The initial value of register  $ScUp_j$  is the same as the initial value of variable  $colors$  of  $Scanner_j$ . The initial value of registers  $H_i$  and  $UpSc_{i,j}$  of  $UpDater_i$  are  $h$  and  $u$  respectively, and are such that  $h.equalities[j]$  is false,  $h.colors[j] = u.color$ , and  $h.parity = p$  is an arbitrary value (0 or 1) such that  $B_i^p = d_0$  is the initial *Data* value.

Figure 7: Initial condition, in case self-stabilization is not an issue.

1. Each  $Scanner_j$ , for  $1 \leq j \leq N$ , writes on a single-writer multiple reader atomic register  $ScUp_j$  (read by each  $UpDater$ ). The values of  $ScUp_j$  are arrays of length  $M$  (number of  $UpDaters$ ) of colors.
2. Process  $UpDater_i$ , for  $1 \leq i \leq M$ , writes on the following registers.
  - (a) Regular registers  $UpSc_{i,j}$  for every  $Scanner$  index  $j$ . Register  $UpSc_{i,j}$  is written by  $UpDater_i$  and read by  $Scanner_j$ . It carries values of type  $UpSc$  (that is two *Color* fields).
  - (b) A multiple reader atomic register  $H_i$  of type *Head* that is read by all *Scanners*.
  - (c)  $UpDater_i$  has  $N + 2$  safe registers (also called buffers) of type *Data*.  $B_{i,j}$  is written by  $UpDater_i$  and is read by  $Scanner_j$ , and  $B_i^0, B_i^1$  are two safe registers written by  $UpDater_i$  and read by all *Scanners*.

Atomicity of a register means serializability of its read/write events. In the correctness proof we shall assume that the atomic registers are already serial. An easy standard argument shows that there is no loss of generality in this assumption (see for example [1]).

The initial values of the local variables and the registers is in Figure 7. Truly, for self-stabilization, any initial state will evolve into a correct behavior and so initial states are of lesser importance. They have not totally lost their place however, since they cause the programs to behave correctly right at the start, without any need for a stabilization period.

We go over the algorithms in order to clarify some points and make some definitions. Following our convention set in the generic algorithm (Figure 6) process  $UpDater_i$  repeats executions of the *UpDate* algorithm, each with some *Data* parameter  $data$ . The set  $U$  of actions (low level events) resulting from an execution of the *UpDate* algorithm is called a high level **update**

*UpDate*(*data*: *Data*) (\* by *UpDater<sub>i</sub>* \*)

1. *parity* := 1 - *parity*;  
write  $B_i^{parity} := data$ ;
2. *old\_colors* := *colors*;
3. forall  $1 \leq j \leq N$ 
  - (a) read *colors*[*j*] := *ScUp<sub>j</sub>*[*i*];
  - (b) write *UpSc<sub>i,j</sub>* :=  $\langle old\_colors[j], colors[j] \rangle$ ;
  - (c) read *c*[*j*] := *ScUp<sub>j</sub>*[*i*];
  - (d) if *colors*[*j*] = *c*[*j*] then *equalities*[*j*] := true  
else *equalities*[*j*] := false;
  - (e) if *equalities*[*j*] and (*colors*[*j*]  $\neq$  *old\_colors*[*j*] or *c*[*j*]  $\neq$  *old\_c*[*j*])  
then write  $B_{i,j} := old\_data$ ;
4. *old\_data* := *data*;  
*old\_c* := *c*;
5. write  $H_i := \langle colors, equalities, parity \rangle$ .

Figure 8: The *UpDate* algorithm for *UpDater<sub>i</sub>* ( $1 \leq i \leq M$ ). Local variables are: *parity*  $\in \{0, 1\}$ ; *colors*, *old\_colors*, *c* and *old\_c* are arrays of length *N* of colors; *equalities* is an array of length *N* of booleans. *data* and *old\_data* are in *Data*. The initial value of variable *old\_data* is  $d_0$ , which is the initial value of the buffers.

*Scan* (\* by *Scanner<sub>j</sub>* \*)

1. forall  $1 \leq i \leq M$ 
  - (a) read  $u[i] := UpSc_{i,j}$ ;
  - (b) let  $colors[i]$  be some *Color* not in  $\{u[i].color, u[i].old\_color, colors[i], h[i].colors[j]\}$ ;
2. write  $ScUp_j := colors$ ;
3. forall  $1 \leq i \leq M$   
read  $b_0[i] := B_i^0$ ; read  $b_1[i] := B_i^1$ ;
4. forall  $1 \leq i \leq M$ 
  - (a) read  $h[i] := H_i$ ;
  - (b) if  $h[i].colors[j] = colors[i]$  and  $h[i].equalities[j]$  then  
read  $values[i] := B_{i,j}$  else  $values[i] := b_{h[i].parity}[i]$
5. return *values*.

Figure 9: The *Scan* algorithm for *Scanner<sub>j</sub>*. Local variables are:  $u$  an array of length  $M$  of *UpSc* values;  $colors$  an array of length  $M$  of colors;  $b_0$ ,  $b_1$ , and  $values$  are arrays of length  $M$  of *Data* values;  $h$  is an array of length  $M$  of *Head* values.

event (or *Update* operation execution), and its parameter *data* is defined to be the value of that event. We write  $Value(U) = data$ . The index  $i$  of the executing process is said to be the index of  $U$ , and we write  $index(U) = i$ . The operation executions by any  $Updater_i$  are linearly ordered, and we say that operation execution  $B$  is the successor of operation execution  $A$  if both are by the same  $Updater_i$ ,  $A \rightarrow B$ , and there is no  $Updater_i$  operation execution  $X$  with  $A \rightarrow X \rightarrow B$ .

If  $x$  is any local variable of  $Updater_i$ ,  $\ell$  is a line number of the code, and  $U$  is an *Update* operation execution by  $Updater_i$ , then  $x^{U,\ell}$  denotes the value of variable  $x$  at  $U$  exactly after executing line  $\ell$ . The value of  $x$  when  $U$  is about to begin execution is denoted  $x^{U,0}$ . The value of variable  $x$  when  $U$  has finished its execution is denoted  $x^U$  (which is the same as  $x^{U,5}$ ). (A similar notation is used for *Scan* executions.)

Line 1 of the *Update* algorithm shows that  $Updater_i$  writes its parameter (*data*) alternately on the safe *Data* registers  $B_i^0$  and  $B_i^1$ .

If  $U_1$  is the successor of  $U_0$  (both are operations by  $Updater_i$ ), then the value of any variable when  $U_0$  has finished its execution is the value of that variable when  $U_1$  begins. So line 2 of the code shows that  $old\_colors^{U_1,2} = colors^{U_0}$ .

Line 3, forall  $1 \leq j \leq N$  (a)—(e), can be executed in any order or by concurrently executing (a)—(e) for  $j$  values 1 to  $N$ . Recall that atomic register  $ScUp_j$  carries an array of length  $M$  of colors. We do not assume that a particular entry,  $ScUp_j[i]$ , can be read directly, and line 3(a)

$$\text{read } colors[j] := ScUp_j[i];$$

is executed by first reading the register and then assigning to  $colors[j]$  the  $i$ th entry of the value read. This reading is done again at line 3(c). In line 3(b), a pair of colors is written back to  $Scanner_j$ : the one just read in executing line 3(a), and the one in the previous execution of that line (if there is a previous *Update* event by this process; in case of the first  $Updater_i$  execution, the initial value of variable *old\_colors* is used, and it can have any value). The array *equalities* is determined in executions of line 3(d).  $equalities[j]$  is set to **true** iff the two reads of  $ScUp_j$  yield the same value at the  $i$ -th entry. In line 3(e),  $Updater_i$  may write the previous *data* value on register  $B_{i,j}$  for  $Scanner_j$  to read. The condition for executing this loading write (as we shall call it) is that the two reads of  $ScUp_j[i]$  yield the same color which is different from one of the values (or value) obtained in the previous  $Updater_i$  operation. In

case of the first  $UpDater_i$  operation execution, the initial values of  $old\_colors$  and  $old\_c$  are used, and these can be any values.

An execution of the  $Scan$  operation by  $Scanner_j$  begins (line 1(a)) by filling variable  $u$  with  $UpSc$  values (which are pairs of colors). The forall instructions in lines 1, 3, and 4 can be executed in any order (or concurrently). In line 1(b) we see why five colors suffice: this allows a choice of a color that is different from all colors in a set of  $\leq 4$  colors. An array,  $colors$ , of “new” colors is established and written in line 2 to all updaters. Then, in line 3, all the  $Data$  safe registers  $B_i^\ell$  are read for  $\ell = 0, 1$ , in any order (no need to first read  $B_i^0$ ). In executions of line 4(a), the scanner reads all  $H_i$  registers and records the  $Head$  values obtained in variable  $h$ . Condition  $h[i].colors[j] = colors[i]$  and  $h[i].equalities[j]$  says, intuitively, that  $UpDater_i$  has succeeded in “covering”  $Scanner_j$  in its two reads of  $ScUp_j$ . In this case register  $B_{i,j}$  is read and its value is returned. We say in this case that the value is returned “indirectly”, because an updater that writes on register  $B_{i,j}$  does not write its present data value but rather the data value of the previous update event (when there is such a previous event). Otherwise, if there is no covering, then one of the values read in line 3 is “directly” returned.

If  $S$  is a  $Scan$  operation execution, then  $Value(S)$  is defined to be the array  $values^S$  which is returned by  $S$ . So  $Value(S)[i]$  is the  $i$ th entry for  $1 \leq i \leq M$ .

## 8 Correctness of the weak snapshot algorithm

Let  $M$  be a normal system execution of the weak snapshot algorithm. Our aim is to prove that  $M$  finally satisfies the stable  $wss$  axioms: for some moment  $m$ ,  $wss(m)$  holds (see Section 6.2). Recall (Section 4) that in an execution of the weak snapshot algorithm there are low level and high level events. We assume a membership predicate  $\in$ , and for events  $a$  and  $B$ ,  $a \in B$  implies that  $a$  is a lower level event that is member of the high level event  $B$ . Lower level events represent read and write actions that correspond to instruction executions. High level events represent the  $Update$  and  $Scan$  operation executions. Normal system executions were defined in 6.2. In a normal execution there exists a moment  $m_0$  such that all crashed processes have terminated before  $m_0$ , the registers operate normally after  $m_0$ , and all non-crashed processes execute  $Scan$  and  $Update$  operations in accordance with their operation programs.

We define a moment  $m'_0 > m_0$  so that each non-crashed  $UpDater_i$  has an *UpDate* operation execution that begins after  $m_0$  and ends before  $m'_0$ . We let  $X^0(UpDater_i)$  be the last such operation execution. Every lower-level event  $e$  that begins after  $m'_0$  belongs to a unique operation execution, denoted  $[e]$ , executed by a non-crashed process.

We also define a moment  $m_1 > m'_0$  so that every non-crashed  $Scanner_j$  process has a *Scan* operation execution that begins after  $m'_0$  and ends before  $m_1$ . We let  $X^0(Scanner_j)$  be the last such *Scan* operation execution. We claim that stabilization occurs after  $m_1$ .

Let  $m$  be a normal moment (that is, by Definition 6.4,  $m \geq m_1$ ). We will prove that  $P_{wss}(m)$  holds. For self-stabilization there is no need for initial conditions, of course, but if self-stabilization is not an issue and processes with their registers behave correctly from start, then the correctness arguments rely on the initial conditions set in Figure 7. In the following proof, however, we concentrate on the stabilization issue, and ask the reader to simplify the proof, when needed, so that it accommodates to the case when it is not question of self-stabilization and weak snapshot should operate properly from the beginning.

Henceforth, in this section, the terms *UpDate* and *Scan* operation executions refer to executions of the protocol by a non-crashed process. Specifically,  $U$  is an *UpDate* operation execution by  $UpDater_i$  if  $UpDater_i$  is non-crashed and  $U$  is  $X^0(UpDater_i)$  or some later event of index  $i$ . Likewise  $S$  is a *Scan* operation execution by  $Scanner_j$  when  $Scanner_j$  is non-crashed and  $S$  is either  $X^0(Scanner_j)$  or a later operation execution by  $Scanner_j$ . An operation execution that begins after  $m$  is said to be a normal operation execution.

Consider the crashed processes in  $M$ . We assume (by normality) that any crashed process contains only events that ended before  $m_0$ , and that it contains write events for each of its registers. After  $m_0$ , any read in  $M$  of a crashed register (that is, a register of a crashed process) will obtain the value of the last write on that register. If  $P$  is a crashed process, we collect all last write events on registers of  $P$  into a higher level event which is defined to be  $X^0(P)$ . (So for a crashed processor  $X^0(P)$  is a high level event that is not an operation execution.) If  $UpDater_i$  is crashed, then this special event is called the ultimate *UpDate* event of index  $i$ , and is denoted  $X^0(UpDater_i)$ . We have in that case that:

1.  $X^0(UpDater_i)$  contains a write on crashed register  $H_i$ , and we let  $h_i^0$

be the value of this crashed register. This is the value returned by all reads of  $H_i$  after  $m_0$ .

2.  $X^0(\text{UpDater}_i)$  contain writes on each register  $\text{UpSc}_{i,j}$  and we let  $u_{i,j}^0$  be the value of that crashed register.
3.  $X_0(\text{UpDater}_i)$  contains writes on its safe *Data* buffers. In fact, we need only the write  $w$  on  $B_i^p$  where  $p = h_i^0.\text{parity}$ . We define the data value of  $X_0(\text{UpDater}_i)$  to be the *Data* value of this write  $w$ . That is  $\text{Value}(X^0(\text{UpDater}_i)) = \text{Value}(w)$ .
4.  $X^0(\text{UpDater}_i)$  ends before  $m_0$ .

If  $\text{Scanner}_j$  is crashed, we let  $X^0(\text{Scanner}_j)$  denote the ultimate *Scan* event of  $\text{Scanner}_j$  in  $M$ . It is a higher-level event that consists of the last write in  $M$  on register  $\text{ScUp}_j$ .

We turn now to the correctness proof. The main issue is to define the  $\Omega$  function, and then to prove that the *wss* intermediate axioms hold after the normal moment  $m$ . Namely that  $P_{wss}(m)$  holds. This will imply that the algorithm self-stabilizes in its implementation of the weak snapshot operations (by Theorem 3.1). We begin with some preliminary definitions and lemmas.

**Definition 8.1 (loading)** *Suppose that  $U$  is an UpDate operation execution of index  $i$ . We say that “ $U$  is in loading mode for index  $j$ ” iff condition  $e(j)$  holds for  $U$ , which is the condition of line 3(e). That is:*

$$\text{equalities}[j] \text{ and } (\text{colors}[j] \neq \text{old\_colors}[j] \text{ or } c[j] \neq \text{old\_c}[j]).$$

*Equivalently,  $U$  is in loading mode for index  $j$  iff  $U$  contains a write on  $B_{i,j}$  (executed in line 3(e)).*

In case  $U$  is the successor of  $U_0$  in  $\text{UpDater}_i$ , and  $U$  is in loading mode for index  $j$ , the following lemma says that the value written by  $U$  on  $B_{i,j}$  (the value loaded) is the data value of  $U_0$ , namely  $\text{Value}(U_0)$ . We say in that case that “ $U$  loads  $U_0$  for index  $j$ ”.

**Lemma 8.2** *Suppose that  $U_0$  and  $U$  are both UpDate events of index  $i$  and  $U$  is the successor of  $U_0$ . Then in an execution of line 3(e) in  $U$  the values of*

variables  $old\_colors$ ,  $old\_c$ , and  $old\_data$  are, respectively,  $colors^{U_0}$ ,  $c^{U_0}$ , and  $Value(U_0)$ .

Let  $j$  be an index and suppose that  $U$  is in loading mode for index  $j$ . Then the data value written on register  $B_{i,j}$  by  $U$  is  $Value(U_0)$ .

This lemma is an example of a statement that holds by virtue of the unrestricted semantics of the algorithm. That is, the properties of the registers employed by  $UpDater_i$  and their correct functioning are irrelevant.

**Definition 8.3 (cover)** 1. An *UpDate* event  $U$  of index  $i$  is said to “cover a color  $s$  in its  $j$ th entry” iff the following holds for  $h$  the value of register  $H_i$  that is written by  $U$ :

$$s = h.colors[j] \text{ and } h.equalities[j].$$

This definition can also be applied in case  $U = X^0(UpDater_i)$  is the ultimate event of a crashed process, because  $U$  contains a write on  $H_i$ .

If  $U$  is an *UpDate* operation execution by a non-crashed process, then since  $U$  writes on  $H_i$  the values of its variables  $colors$  and  $equalities$  in the corresponding fields, we could say alternatively that  $U$  covers  $s$  in its  $j$ th entry iff

$$s = colors^U[j] \text{ and } equalities^U[j].$$

For a crashed process, however, the local variables are insignificant and the definition of covering relies on the final write on the  $H_i$  register of the crashed process.

2. An *UpDate* event  $U$  of index  $i$  is said to “cover” a *Scan*  $S$  of index  $j$  iff  $U$  covers  $colors^S[i]$  in its  $j$ th entry, and for  $t \in U$  that is the second read in  $U$  of register  $ScUp_j$  we have  $\omega(t) \in S$ .

A simple situation in which  $U$  covers  $S$  is when both reads in  $U$  of the  $ScUp_j$  register of  $S$  obtain the write by  $S$ . It follows easily that if  $S$  is any *Scan* operation then the set of *UpDate* operations of index  $i$  that cover  $S$  is convex: If  $U_0 \rightarrow U_1 \rightarrow U_2$  are all *UpDate* operations in  $UpDater_i$  and both of  $U_0$  and  $U_2$  cover  $S$ , then  $U_1$  covers  $S$  as well (recall that  $ScUp_j$  is assumed to be a serial register).

**Definition 8.4** *The set of all UpDate events in UpDater<sub>i</sub> that cover S is called the covering block of S in UpDater<sub>i</sub> (it may well be empty).*

Each Scan operation S avoids in its choice of colors[i] the two colors obtained in reading register UpSc<sub>i,j</sub>. The following lemma is an immediate consequence of this behavior.

**Lemma 8.5** *Suppose that S is a Scan operation by Scanner<sub>j</sub>, and r is the read of UpSc<sub>i,j</sub> done in line 1(a) in S. Suppose that the write ω(r) is in an UpDate event U. If s = colors<sup>S</sup>[i] is the color determined by S in executing line 1(b), then U does not cover s in its jth entry. Moreover, if U has a predecessor U<sub>0</sub> in UpDater<sub>i</sub> then U<sub>0</sub> does not cover s in its jth entry.*

Each Scan operation S avoids in its choice in line 1(b) the color of variable colors[i] which is the color written on ScUp<sub>j</sub> in its ith entry by the previous operation. This entails the following obvious lemma.

**Lemma 8.6** *Let S<sub>0</sub> and S<sub>1</sub> be a Scan operation and its successor in Scanner<sub>j</sub>, and suppose that V is an UpDate operation of index i. Let t ∈ V be either the first or the second read of register ScUp<sub>j</sub>, and suppose that [ω(t)] = S<sub>0</sub>. Then V does not cover colors<sup>S<sub>1</sub></sup>[i] in its jth entry.*

*Proof.* Since ω(t) ∈ S<sub>0</sub>, t obtains the value colors<sup>S<sub>0</sub></sup>. Now S<sub>1</sub> determines in line 1(b) a new value s for colors[i] (namely colors<sup>S<sub>1</sub></sup>[i]) that is different from colors<sup>S<sub>0</sub></sup>[i]. So either equalities<sup>V</sup>[j] does not hold (and then certainly V does not cover s in its jth entry) or else we have colors[j] = c[j] = colors<sup>S<sub>0</sub></sup>[i] in V, and hence s ≠ colors<sup>U</sup>[j]. ■

The following lemma formalizes our intuitive understanding of the loading mechanism. When U is an UpDater<sub>i</sub> event that covers color s in its jth entry, then U “presumes” that Scanner<sub>j</sub> may need an older value and a loading of register B<sub>i,j</sub> is needed. If, however, the previous UpDate operation also covers s, then U is no longer responsible for that loading and therefore no loading occurs in U in this case.

**Lemma 8.7** *Let s be a color, 1 ≤ j ≤ N an index, and let U and U<sub>0</sub> be UpDate events such that U is the successor of U<sub>0</sub> in UpDater<sub>i</sub>, and U covers s in its jth entry. Then U is in loading mode for j if and only if U<sub>0</sub> does not cover s in its jth entry.*

*Proof.* Since  $U$  covers  $s$  in its  $j$ th entry,  $equalities^U[j]$  holds and thus

$$s = colors[j] = c[j] \tag{3}$$

holds in  $U$  at line 3(c).

Assume first that  $U$  is in loading mode for  $j$ . That is, condition  $e(j)$  (from Definition 8.1) holds for  $U$ . Thus  $U$  finds at line 3(e) that ( $s \neq old\_colors[j]$  or  $s \neq old\_c[j]$ ). It follows from Lemma 8.2 above that in executing line 3(e)  $U_0$  finds that ( $s \neq colors[j]$  or  $s \neq c[j]$ ). This implies that  $U_0$  does not cover  $s$  in its  $j$ th entry.

Assume now that  $U$  is not in loading mode for  $j$  and hence that condition  $e(j)$  does not hold for  $U$ . Since  $equalities^U[j]$  holds,  $colors[j] = old\_colors[j]$  and  $c[j] = old\_c[j]$  hold in  $U$  (in executing line 3(d)). This implies that  $colors[j] = c[j]$  holds in  $U_0$  as well (by 3) and thus it follows that  $U_0$  covers  $s$  in its  $j$ th entry. ■

**Lemma 8.8** *Suppose that  $U_0 \rightarrow U$  are an UpDate and its successor in UpDater <sub>$i$</sub> , and  $S$  is a normal Scan event of index  $j$ . Let  $s = colors^S[i]$ . If  $U$  covers  $S$  and  $U_0$  covers  $s$  in its  $j$ th entry, then  $U_0$  covers  $S$ .*

**Proof:** Let  $r_2 \in U_0$  and  $t_2 \in U$  be the second reads of the serial register  $ScUp_j$  in these operations. Let  $u_0 \in U_0$  and  $u \in U$  be the write actions there on register  $UpSc_{i,j}$ . We have  $u_0 \rightarrow r_2 \rightarrow u \rightarrow t_2$ . Let  $q \in S$  be the read of register  $UpSc_{i,j}$ , and  $w \in S$  be the write on register  $ScUp_j$ . As  $U$  covers  $S$  we have  $w = \omega(t_2)$ . If  $w \rightarrow r_2$  then  $w = \omega(r_2)$  would follow (from the atomicity of the  $ScUp_j$  register) and imply that  $U_0$  covers  $S$ . Assume now that

$$r_2 \rightarrow w.$$

Let  $S_0$  be the immediate predecessor of  $S$  in  $Scanner_j$  (since  $S$  is normal,  $X^0(Scanner_j) \rightarrow S$ , and hence  $S_0$  exists, a Scan operation execution). Let  $w_0 \in S_0$  be the write on  $ScUp_j$ . Lemma 8.6 above says that  $\omega(r_2) \in S_0$  is impossible (it would imply that  $U_0$  does not cover  $s$  in its  $j$ th entry). So  $r_2 \rightarrow w_0$  and  $u_0 \rightarrow q$  follow. We conclude that  $q$  obtains either the write in  $U_0$  or the write in  $U$  (because it is contained in the interval  $(u_0, t_2)$ ). In either case, Lemma 8.5 implies that  $U_0$  does not cover  $s$  in its  $j$ th entry, and we get a contradiction. ■

Together with Lemma 8.7, and Lemma 8.2, Lemma 8.8 yields the following. (The covering block is defined in 8.4.)

**Corollary 8.9** *Let  $S$  be a normal Scan operation of index  $j$ , and let  $U$  be an UpDate operation of index  $i$  that is the first operation in the covering block of  $S$  (in the  $\rightarrow$  ordering). Suppose that  $U_0$  is the predecessor of  $U$  in  $UpDater_i$ . Then  $U$  is in loading mode for index  $j$ , and it loads  $U_0$  for that index. It also follows from Lemma 8.7 that  $U_0$  does not cover  $colors^S[i]$  in its  $j$ th entry.*

We now define an auxiliary function  $\Gamma(S, i)$  for every normal Scan  $S$  (namely one that begins after  $m$ ) and  $UpDater$  index  $i$ . Let  $r$  be the read of  $H_i$  in  $S$  (corresponding to line 4(a)). Then define  $\Gamma(S, i) = [\omega(r)]$ . (Recall from section 4 that for a read event  $r$  on register  $H_i$ ,  $\omega(r)$  is the corresponding write on that register, and  $[\omega(r)]$  is the higher level event containing that write.)

If  $UpDater_i$  is crashed then  $\Gamma(S, i) = X^0(UpDater_i)$  is the ultimate  $UpDater_i$  event. In case  $UpDater_i$  is non-crashed,  $m'_0 \rightarrow r$  implies that  $X^0(UpDater_i) \rightarrow r$  and hence  $X^0(UpDater_i) = \Gamma(S, i)$  or  $X^0(UpDater_i) \rightarrow \Gamma(S, i)$ . In any case,  $\Gamma(S, i)$  is either a justified  $UpDater_i$  event that begins after  $m_0$ , or the ultimate crashed event of  $UpDater_i$  if that process is crashed.

The following is an immediate consequence of the seriality of the *Head* register  $H_i$ .

**Lemma 8.10** *For every normal Scan  $S$ ,  $end(\Gamma(S, i)) < end(S)$ . In fact, if  $r \in S$  is the read of register  $H_i$  then  $\Gamma(S, i) \rightarrow r$ .*

**Definition 8.11** *Let  $1 \leq i \leq M$  be an index. A normal Scan operation  $S$  of index  $j$  is said to be in “direct mode for  $i$ ” iff the following holds in  $S$  in the execution of line 4(a):*

$$h[i].colors[j] \neq colors[i] \text{ or } h[i].equalities[j] = \text{false}.$$

That is, in executing 4(b) for index  $i$ , the condition of the if statement does not hold and  $S$  determines  $values[i] := b_p[i]$  (where  $p = h^S[i].parity$ ).

**Lemma 8.12** *For a normal Scan event  $S$  of index  $j$ , and for any  $1 \leq i \leq M$ ,  $S$  is in direct mode for  $i$  iff*

$$\Gamma(S, i) \text{ does not cover } colors^S[i].$$

**Proof.** Say  $U = \Gamma(S, i)$ . By definition,  $U$  does not cover  $colors^S[i]$  iff

$$colors^U[j] \neq colors^S[i] \text{ or } \neg equalities^U[j]. \quad (4)$$

Let  $r$  be the read of  $H_i$  in  $S$ . By definition,  $U = [\omega(r)]$ , and hence the value written by  $U$  on  $H_i$  is the value returned by  $r$  (which is assigned to  $h[i]$ ). Thus

$$h[i]^S = \langle colors, equalities, parity \rangle^U,$$

so that  $h^S[i].colors = colors^U$ ,  $h^S[i].equalities = equalities^U$ . Hence (4) is equivalent to

$$h^S[i].colors[j] \neq colors^S[i] \text{ or } \neg h^S[i].equalities[j]$$

But this is exactly the statement “ $S$  is in direct mode for  $i$ ”, which proves the lemma. ■

**Lemma 8.13** *Let  $S$  be a normal Scan event, and  $1 \leq i \leq M$  an index such that  $UpDater_i$  is crashed. Then  $X^0(UpDater_i)$  does not cover  $colors^S[i]$ , and  $S$  is in direct mode for  $i$ . In this case,  $Value(S)[i] = Value(X^0(UpDater_i))$ .*

**Proof:** Let  $j$  be the index of  $S$ . By definition,  $S$  is normal means that  $S$  begins after moment  $m$ . So  $X^0(Scanner_j) \rightarrow S$ . It follows that  $S$  has a predecessor  $S_0$  in  $Scanner_j$  that is a justified execution of the *Scan* protocol. Clearly the read of register  $H_i$  in  $S_0$  obtains the write in  $X = X^0(UpDater_i)$  (the ultimate event of the crashed process  $UpDater_i$ ). It follows that when  $S$  executes line 1(b) for index  $i$  and chooses a color not in  $\{u[i].color, u[i].old\_color, colors[i], h[i].colors[j]\}$ , the variable  $h[i].colors[j]$  that it considers is  $h^{S_0}[i]$ , namely the value obtained by  $S_0$  in reading  $H_i$ . This implies that  $X = \Gamma(S, i)$  does not cover  $colors^S[i]$ . Hence  $S$  is in direct mode for  $i$ . It now follows that  $values^S[i]$  is the value of  $b_{h[i].parity}$  which implies  $Value(S)[i] = Value(X^0(UpDater_i))$ , since  $Value(X^0(UpDater_i))$  is by definition the value of the last write on buffer  $b_{h[i].parity}$ . ■

**Lemma 8.14** *Let  $S$  be a normal Scan operation of index  $j$ . Let  $i$  be an updater index and suppose that  $s = colors^S[i]$ . If  $U = \Gamma(S, i)$  covers  $s$  in its  $j$ th entry, then  $U$  covers  $S$ . Hence, by definition,  $U$  is in the covering block of  $S$ .*

**Proof.** Let  $w \in S$  be the write on register  $ScUp_j$ , and  $t_2 \in U$  be the second read of that register (executed for line 3(c)); we must prove that  $w = \omega(t_2)$ , and we first prove that  $w \rightarrow t_2$ . Since  $U$  covers  $s$ ,  $UpDater_i$  is not crashed (by Lemma 8.13), and  $U$  is a justified  $UpDater_i$  operation execution in  $M$ .

In particular,  $t_2 \in U$  exists. Suppose that  $t_2 \rightarrow w$  (as  $ScUp_j$  is assumed to be serial, this is the alternative option). Consider  $\omega(t_2) \rightarrow w$ . It is not possible that  $\omega(t_2)$  belongs to the predecessor of  $S$ , since  $S$  avoids its previous color. (That is,  $colors^S[i] \neq colors^{S_0}[i]$ . Note also that since  $S$  is normal it has a predecessor in  $Scanner_j$  which is an operation execution that begins after  $m_0$ .) So, if  $S_0$  is the predecessor of  $S$  in  $Scanner_j$  and  $w_0 \in S_0$  is the write on  $ScUp_j$ , then  $t_2 \rightarrow w_0$ , and so  $t_2 \rightarrow S$ . Consider now  $s \in S$ , the read of  $UpSc_{i,j}$ . Let  $u \in U$  be the write on  $UpSc_{i,j}$ . Since  $u \rightarrow t_2$ ,  $u \rightarrow s$ . Thus  $u \rightarrow \omega(s)$  or  $u = \omega(s)$ . Since  $U$  covers  $s$ ,  $u = \omega(s)$  is impossible. But  $\omega(s) \in U^+$  is also impossible when  $U^+$  is the successor of  $U$  in  $UpDater_i$ . Hence  $U^+ \rightarrow \omega(s)$ . This entails  $U^+ \rightarrow r$  where  $r \in S$  is the read of  $H_i$ . But this contradicts  $U = \Gamma(S, i)$ .

Hence  $w \rightarrow t_2$ , and since  $U = \Gamma(S, i)$ ,  $end(t_2) < end(S)$  follows immediately which implies that  $w = \omega(t_2)$ . ■

Next we define the main function  $\Omega$  in two cases. For any normal *Scan* event  $S$  and *UpDater* index  $1 \leq i \leq M$ :

1. If  $S$  is in direct mode for  $i$  define  $\Omega(S, i) = \Gamma(S, i)$ . In particular, if  $UpDater_i$  is crashed then  $\Omega(S, i) = X^0(UpDater_i)$  is the ultimate event of  $UpDater_i$  in  $M$ .
2. If  $S$  is in indirect mode for  $i$  then  $U = \Gamma(S, i)$  covers  $S$  (by lemmas 8.12 and 8.14), and hence the covering block of  $S$  in  $UpDater_i$  is non-empty. Let  $U_1$  be the first operation in that block. Since  $X^0(UpDater_i) \rightarrow S$ , it is not the case that  $X^0(UpDater_i)$  covers  $S$ , and hence  $X^0(UpDater_i) \rightarrow U_1$  and  $U_1$  has a predecessor  $U_0$  in  $UpDater_i$ . The minimality of  $U_1$  implies that  $U_0$  is not in the covering block of  $S$ . We define in this case  $\Omega(S, i) = U_0$ .

**Lemma 8.15** *For every normal Scan operation  $S$  of index  $j$ , if  $r \in S$  is the read of register  $H_i$  then:*

1.  $\Omega(S, i) \rightarrow r$ ,
2.  $\Omega(S, i)$  does not cover  $colors^S[i]$  in its  $j$ th entry.

**Proof:** We know that  $\Gamma(S, i) \rightarrow r$  (Lemma 8.10) and as  $\Omega(S, i)$  is either  $\Gamma(S, i)$  or an earlier operation,  $\Omega(S, i) \rightarrow r$  follows.

That  $\Omega(S, i)$  does not cover  $colors^S[i]$  in its  $j$ th entry can be checked by following the two cases in the definition of  $\Omega(S, i)$ . In case  $S$  is in direct mode

for  $i$ , we defined  $\Omega(S, i) = \Gamma(S, i)$ , and then Lemma 8.12 implies that  $\Omega(S, i)$  does not cover  $colors^S[i]$  in its  $j$ th entry. In case  $S$  is in indirect mode for  $i$ , we follow the definition of  $\Omega(S, i)$  and use Corollary 8.9. ■

At this stage, we do not know yet that each normal *Scan*  $S$  returns the value of  $\Omega(S, i)$  in its  $i$ -th entry. The problem is that the values returned by  $S$  are those determined by reads of the buffers (while  $\Omega$  is determined without any direct reference to the buffers). The following two lemmas check the cases that  $S$  is in direct and then in indirect mode for  $i$  and show in both cases that

$$Value(S)[i] = Value(\Omega(S, i)).$$

**Lemma 8.16** *Suppose that normal Scan  $S$  of index  $j$  is in direct mode for  $i$ . Let  $b \in S$  be the read of register  $B_i^p$ , where  $p = h^S[i].parity$ . Then  $[\omega(b)] = \Gamma(S, i)$  and  $Value(S)[i] = Value(\Omega(S, i))$ .*

**Proof.** The case when  $UpDater_i$  is crashed has already been dealt with in Lemma 8.13, so we assume that this process is non-crashed. Say  $U = \Gamma(S, i)$ , and let  $r \in S$  be the read of  $H_i$ . So  $U = [\omega(r)]$  is in  $UpDater_i$ . Let  $u \in U$  be the write on buffer  $B_i^p$ . We will first prove that  $u \rightarrow b$ . Then we will prove that if there is another write  $u^+$  on  $B_i^p$  that follows  $u$  then  $b \rightarrow u^+$ . Thus  $b$  is not concurrent with any write on this safe buffer, and hence  $u = \omega(b)$  and  $Value(u) = Value(b)$  concludes the proof of the lemma. Assume that  $u \rightarrow b$  does not hold. Then we have  $begin(b) \leq end(u)$ , and this entails that the two reads in  $U$  of register  $ScUp_j$  are in the interval  $(begin(b), begin(r))$ . Thus,  $U$  covers  $S$ , which is not the case (by Lemma 8.12).

Now assume that  $UpDater_i$  has another write  $u^+$  on  $B_i^p$  such that  $u \rightarrow u^+$ . Since  $UpDater_i$  writes alternately on registers  $B_i^0$  and  $B_i^1$ , there has to be an operation execution  $V$  by  $UpDater_i$  that is the successor of  $U$  and such that  $U \rightarrow V \rightarrow u^+$ . Since  $U \rightarrow V \rightarrow r$  is impossible,  $begin(r) \leq end(V)$ , and hence  $b \rightarrow r \rightarrow u^+$  follows. ■

**Lemma 8.17** *If normal scan  $S$  is in indirect mode for  $i$ , then  $Value(S)[i] = Value(\Omega(S, i))$ .*

**Proof.** Assume that *Scan*  $S$  of index  $j$  is in indirect mode for  $i$ , and let  $U = \Gamma(S, i)$ . So  $U$  covers  $S$  by Lemma 8.12. Hence the covering block of  $S$  in  $UpDater_i$  is nonempty. Recall the definition of  $\Omega(S, i)$  in this case. We first defined  $U_1$  as the first member of that covering block, and then defined

$U_0 = \Omega(S, i)$  as the immediate predecessor of  $U_1$  in  $UpDater_i$ . As  $U_0$  is not in the covering block of  $S$ ,  $U_0$  does not cover  $s = colors^S[i]$  in its  $j$ th entry (Lemma 8.8) and hence  $U_1$  is in loading mode for index  $j$  (Lemma 8.7) and it loads  $U_0$  for index  $j$  (Lemma 8.7). That is,  $U_1$  contains a write  $f_1$  on  $B_{i,j}$  and the value of this write is  $Value(\Omega(S, i))$ . We must prove that the read  $b \in S$  of  $B_{i,j}$  (which exists since  $S$  is in indirect mode for  $i$ ) obtains the write  $f_1$ . That is,  $f_1 = \omega(b)$ . Since  $\Gamma(S, i)$  is in the covering block of  $S$  in  $UpDater_i$ ,  $U_1 \rightarrow \Gamma(S, i)$  or  $U_1 = \Gamma(S, i)$  follows the minimality of  $U_1$ . Hence  $U_1 \rightarrow r$ , where  $r \in S$  is the read of  $H_i$ , and  $U_1 \rightarrow b$  follows (from  $r \rightarrow b$ ). We will prove next that if  $f_2$  is any write on buffer  $B_{i,j}$  that follows  $f_1$  then  $b \rightarrow f_2$ . So that  $f_1 \rightarrow b \rightarrow f_2$ , and  $f_1 = \omega(b)$  follows from the assumed safeness of  $B_{i,j}$ . Suppose  $U_2$  is an  $UpDate$  operation of index  $i$  and  $f_2 \in U_2$ . So  $U_1 \rightarrow U_2$ . If  $\neg(b \rightarrow f_2)$  then  $begin(f_2) \leq end(b)$  and it follows then that  $U_2$  covers  $S$ . But then  $U_2$  is in the covering block of  $S$  and it is not in loading mode for  $j$  since it is not the first operation in that block. ■

**Lemma 8.18** *For every Scan event  $S$  and  $UpDater$  index  $i$  there is no  $UpDater_i$  operation  $V$  such that  $\Omega(S, i) \rightarrow V \rightarrow S$ . In fact, if  $w \in S$  is the write on  $ScUp_j$ , then  $\Omega(S, i) \rightarrow V \rightarrow w$  is impossible.*

**Proof.** Assume on the contrary that  $\Omega(S, i) \rightarrow V \rightarrow w$ , where  $V$  is an  $UpDate$  of index  $i$  and  $w \in S$  is the write on  $ScUp_j$ . Let  $r \in S$  be the read of register  $H_i$ . Since  $V \rightarrow w \rightarrow r$ ,  $V \rightarrow r$ , and hence  $\Gamma(S, i)$  is either  $V$  or a later operation by  $UpDater_i$ . Anyhow,  $\Omega(S, i) \rightarrow \Gamma(S, i)$  implies that  $S$  is in indirect mode for index  $i$  and  $\Omega(S, i)$  is defined as the predecessor of the first operation in the covering block of  $S$  in  $UpDater_i$ . But this is impossible, since the covering block is convex and  $V$  is certainly not in that block (it doesn't cover  $S$  as it precedes  $w$ ). ■

If we try to prove the stable *wss* intermediate axioms, we will see that axiom  $2_m(b)$  (of Figure 5) is not necessarily true: if  $U = \Omega(S, i)$  then there is no reason for  $begin(U) < begin(S)$  to hold. Scan  $S$  begins with a series of read actions some of which may even precede  $U$ . We shall define, however, for every Scan event  $S$  a sub-operation  $[S]$  which we call “reduced Scan” and prove the *wss* intermediate axioms for the resulting higher level events (reduced scans, update events, and ultimate events). Then by Theorem 3.1 the *wss* properties hold for these events. An easy argument would then yield that the original events (with the fuller scans) also satisfy the *wss* properties, and this proves that the algorithm implements a weak snapshot system.

**Definition 8.19 (Reduced Scans)** For any Scan operation execution  $S$  of index  $j$  let  $\lfloor S \rfloor$  be the set of actions of  $S$  that includes its write on  $ScUp_j$  and the subsequent reads.

Thus  $\lfloor S \rfloor$  is obtained from  $S$  by removing all the reads of registers  $UpSc_{i,j}$ . For notational uniformity, it is convenient to allow the application of the  $\lfloor \cdot \rfloor$  functional to  $UpDate$  events as well, and to define  $\lfloor U \rfloor = U$  for any  $UpDate$  event  $U$ . So a “reduced” update is just an update event.

We extend our functions and define  $\Omega(\lfloor S \rfloor, i) = \Omega(S, i)$  and  $Value(\lfloor S \rfloor) = Value(S)$ .

**Lemma 8.20** If  $U = \Omega(\lfloor S \rfloor, i)$  then  $begin(U) < begin(\lfloor S \rfloor)$ .

**Proof.** Let  $w \in S$  be the write on  $ScUp_j$ , let  $r \in S$  be the read of  $H_i$ , and let  $t_1 \in U$  be the first read of  $ScUp_j$ . We will prove that  $t_1 \rightarrow w$ , which implies the lemma. If this is not the case, then  $w \rightarrow t_1$  by the seriality of  $ScUp_j$ . But then the two reads of  $ScUp_j$  in  $U$ ,  $t_1$  and  $t_2$ , are in the interval  $(end(w), begin(r))$ , and it follows that  $U$  covers  $S$ , in contradiction to Lemma 8.15. ■

The *wss* intermediate axioms are now obvious for the reduced scans and updates. In particular axiom 2(b) holds by the previous lemma, and axiom 2(c) by Lemma 8.18).

Hence the *wss* properties hold for the reduced operation executions, and we have to argue that they also hold for the original ones. This is almost trivial. There is a partial ordering  $\Rightarrow^R$  defined on the reduced operation so that the *wss* properties hold. We define the “same” relation on the original operations. That is, we define  $A \Rightarrow B$  iff  $\lfloor A \rfloor \Rightarrow^R \lfloor B \rfloor$ . To prove that the  $\Rightarrow$  relation extends the  $\rightarrow$  relation, assume that  $A \rightarrow B$  holds for (regular) event  $A$  and  $B$ . But then clearly  $\lfloor A \rfloor \rightarrow \lfloor B \rfloor$ . Hence  $\lfloor A \rfloor \Rightarrow^R \lfloor B \rfloor$ . By definition this implies  $A \Rightarrow B$ .

## 8.1 Selective Scans

There are situations in which a *Scan* operation is not required to read shared memory variables of all the updaters, but only a selected subset of them. We shall see an example of this in the second part of our paper. If  $R_1, \dots, R_M$  are the names of the registers of  $UpDater_1, \dots, UpDater_M$ , then, for any subset  $i_1, \dots, i_k$  of indices, operation

$$Scan(R_{i_1}, \dots, R_{i_k})$$

returns a weak snapshot view of just those  $k$  variables specified in the call. The *UpDate* operation does not change, but the *Scan* protocol is changed and instead of the **forall**  $1 \leq i \leq M$  instructions of lines 1,3, and 4, we would have **forall**  $i_\ell$  where  $1 \leq \ell \leq k$ .

## 9 Timestamps

Here begins the second part of the paper in which an efficient self-stabilizing timestamping protocol is described. We follow the main idea and the exposition of [5], but instead of using atomic registers as in that paper, we use the *Scan/UpDate* operations that implement weak snapshots as developed in the first part of the present paper. Since [5] already contains illustrations and informal explanations of some of the main ideas, we allow here a more succinct presentation. Our approach to the specification of timestamps is almost the same as that of Dolev and Shavit [14], and section 2.6.1 in [5] discusses and explains the differences.

There is a minor terminology problem that we must first straighten up. The operations of weak snapshots are called update and scan, and the operations of timestamping are called label and scan. So the term “scan” is used both for snapshots and for timestamps and this may create some confusion. Instead of inventing a new term we shall use *UpDate* and *Scan* for the weak snapshot operations, and use *LABEL* and *SCAN* for our timestamps. Upper case *S* will denote *SCAN* operations, and lower case *s* will denote *Scan* operations.

### 9.1 The timestamp specification language

We shall define here the first-order language for specifying concurrent timestamp systems, denoted  $L_{TS}$ . It is a system execution signature that consists of the following constants, functions, and predicates (in addition to those symbols such as  $\rightarrow$ , *begin*, *end*, *Moment*, and  $<$  which are common to every system execution; see Definition 2.1).

**atemporal sorts:** Sorts *LabelerIndex* and *message\_type*. *LabelerIndex* contains the indices of the labeling processes, and we have constants  $1, \dots, N$  to denote the members of *LabelerIndex*. Sort *message\_type* represents the class of all possible messages. (Communication here is with shared

memory, not message passing, but we use the term “messages” for the data values that the labelers write and scanners read.)

An additional sort is needed, the collection of all finite sequences needed to represent linear orderings of *LabelerIndex* and sequences of length  $N$  of messages.

**unary functions:** *index*,  $\triangleleft$ , *message*, *messages*.

**unary predicates:** *SCAN* and *LABEL*. If *LABEL*( $L$ ) and *index*( $L$ ) =  $i$  then we say that  $L$  is a labeler event of index  $i$ , and we write this for short as *Labeler<sub>i</sub>*( $L$ ). If *SCAN*( $S$ ), we say that  $S$  is a scan event, or operation. In this case *messages*( $S$ ) is a sequence of messages in *message\_type*, and we write *messages*( $S$ )( $i$ ) for the  $i$ -th entry in this sequence. (It is supposed to represent the message captured from *Labeler<sub>i</sub>* and returned by  $S$ .) For any *Labeler* event  $L$ , *message*( $L$ ) is some value in *message\_type*. We say that *message*( $L$ ) is the message written by  $L$ . For any any *SCAN* event  $S$ ,  $\triangleleft(S)$  is an ordering of *LabelerIndex*.

**binary predicate and function:**  $\Rightarrow$  is a binary predicate on the set of *Labeler* events. (It is used as the global ordering of the set of labeling operation executions.) The reader may recall that we used this symbol for the *wss* list of properties, but in this section we shall use  $\Rightarrow$  only on the labeling events, so that this will not be a source of confusion.

$\Omega$  is a binary function: if *SCAN*( $S$ ) and *LabelerIndex*( $i$ ), then  $\Omega(S, i)$  is a *Labeler<sub>i</sub>* event. We have used the symbol  $\Omega$  also in the weak snapshot specification, but, again, this overloading will not be a source of confusion, especially since we shall write  $\Omega_i(S)$  rather than  $\Omega(S, i)$  when  $S$  is a *SCAN*, and reserve  $\Omega(s, i)$  to *Scan* events  $s$ .

Intuitively,  $\Omega_i(S) = M$  means that the *SCAN* event  $S$  “has seen and returned” the (value of the) *Labeler<sub>i</sub>* event  $M$ .

This ends the description of the symbols and hence of a first-order language for timestamp systems which we denote  $L_{TS}$ . The following subsection contains a list of requirements (called axioms) formulated as sentences of this language, and any system execution that satisfies these sentences is called a *cts* execution (Concurrent TimeStamp system execution).

## 9.2 Definition of concurrent timestamps

This subsection contains a list of statements, called *cts* axioms, in the language  $L_{TS}$ . This yields a formal definition of the correct behavior of timestamping protocols.

**Definition 9.1** *The cts axioms are the following:*

**processes:** The processes are pairwise disjoint. That is,

$$\forall x, y (SCAN(x) \wedge LABEL(y) \longrightarrow x \neq y)$$

and likewise for every *LabelerIndex*  $i \neq j$

$$\forall x, y (Labeler_i(x) \wedge Labeler_j(y) \longrightarrow x \neq y).$$

Each *Labeler* process is serial: for every  $i$  such that *LabelerIndex*( $i$ )

$$Labeler_i(a) \wedge Labeler_i(b) \wedge a \neq b \longrightarrow (a \rightarrow b \vee b \rightarrow a).$$

(It would have been possible to require that the set of *SCAN* events is partitioned into serial processes, however this is not necessary and *Scanner* is viewed here as a single process which is not necessarily serial.)

**regularity:** For each *LabelerIndex*  $i$ ,  $\Omega_i$  is a regular function from *Scanner* to *Labeler<sub>i</sub>*. That is for every *SCAN*  $S$ :

1.  $\Omega_i(S) \not\rightarrow S$  (or equivalently  $begin(S) < end(\Omega_i(S))$ ), and
2. there is no *Labeler<sub>i</sub>* event  $W$  such that  $\Omega_i(S) \rightarrow W \rightarrow S$ .

**Bloom's property:** For every *SCAN* event  $S$  and *LabelerIndex*  $i$

$$end(\Omega_i(S)) < end(S).$$

(Bloom [13] used such a property in a different context.)

**global ordering:**  $\Rightarrow$  is a linear order on the *LABEL* events which extends  $\rightarrow$ . That is, if  $b_1$  and  $b_2$  are *LABEL* events such that  $b_1 \rightarrow b_2$  then  $b_1 \Rightarrow b_2$ .

**local ordering:** For every *SCAN* event  $S$ ,  $\triangleleft(S)$  is an irreflexive ordering of *LabelerIndex*, and  $messages(S)$  is a sequence of *message\_type* indexed by *LabelerIndex*.

**global coherence:** For every *SCAN* event  $S$  and indexes  $i \neq j$ , if  $L = \Omega_i(S)$  and  $M = \Omega_j(S)$  then:

$$i \triangleleft(S) j \text{ iff } L \Rightarrow M.$$

Moreover,  $message(\Omega_i(S)) = messages(S)(i)$  for every *LabelerIndex*  $i$ .

We are going to describe in the sequel protocols that implement operations *LABEL*, and *SCAN* so that any system execution self-stabilizes to the *cts* axioms described above. The precise definition of self-stabilization of concurrent timestamps is given in the following.

### 9.2.1 Definition of stabilizing *cts*

If  $H$  is a system execution that interprets the  $L_{TS}$  language, it is possible that the *cts* axioms do not hold in  $H$  but they do after a certain moment  $m$ . In this case we shall say that the axioms *finally* hold in  $H$ , or that  $H$  finally satisfies the *cts* axioms. For a formal definition of this concept we follow the treatment of Section 6, and define a property  $P_{cts}(m)$  which says, intuitively, that the events that begin after moment  $m$  satisfy the *cts* axioms.

One of the main issues in this definition is to decide for a *SCAN* event  $S$  that begins after  $m$  what should  $\Omega_i(S)$  represent when *Labeler<sub>i</sub>* is crashed. One possibility is to require that  $\Omega_i(S)$  is defined only when *Labeler<sub>i</sub>* is non-crashed. For a crashed *Labeler<sub>i</sub>* one would require then that there is a fixed value which is returned by every *SCAN* after  $m$  in its  $i$ -th entry. Another possibility, which we adopt since it is more succinct, is to assume that every crashed *Labeler<sub>i</sub>* has an “ultimate” event  $U_i$  such that for every *SCAN*  $S$  after  $m$   $U_i = \Omega_i(S)$  holds. This will immediately imply that any *SCAN* after  $m$  returns the value of  $U_i$  in its  $i$ -th entry.

**Definition 9.2** *Let  $m$  be a variable of sort Moment. Let  $Y_m$  be the collection of all events in  $H$  that begin after  $m$ . Property  $P_{cts}(m)$  is the following conjunction which is, essentially, obtained from the *cts* axioms by restricting the universal quantifiers to the events in  $Y_m$ .*

**processes:** The processes are pairwise disjoint, and each process is serially ordered.

**ultimate events:** If  $Labeler_i$  is crashed, then it contains an ultimate event  $U_i$ , it is the last event of  $Labeler_i$ , and  $end(U_i) < m$ .

**regularity:** For each  $LabelerIndex$   $i$ , the restriction of  $\Omega_i$  to  $Y_m$  is a regular function. That is for every  $SCAN$   $S$ , if  $m < begin(S)$  then

1.  $S \not\rightarrow \Omega_i(S)$  and
2. there is no  $Labeler_i$  event  $W$  such that  $\Omega_i(S) \rightarrow W \rightarrow S$ .

(We do not require that  $\Omega_i(S)$  begins after  $m$ . Item 2 excludes any  $W$  such that  $\Omega_i(S) \rightarrow W \rightarrow S$ , including such  $W$  that begin before  $m$ .)

**Bloom's property:** For every  $SCAN$  event  $S$  that begins after  $m$  and for every  $LabelerIndex$   $i$

$$end(\Omega_i(S)) < end(S).$$

**global ordering:**  $\Rightarrow$  is a linear ordering of the set of all events of the form  $\Omega_i(S)$  where  $S$  is a  $SCAN$  event such that  $m < begin(S)$  (and in this case  $\Omega_i(S)$  is a *LABEL* event or the ultimate event of  $Labeler_i$  if  $Labeler_i$  is crashed). If  $A$  and  $B$  are in the domain of  $\Rightarrow$ , then  $A \rightarrow B$  implies  $A \Rightarrow B$ , except when both  $A$  and  $B$  are ultimate events (in which case the temporal ordering is immaterial).

**local ordering:** For every  $SCAN$  event  $S$ , if  $m < begin(S)$  then  $\triangleleft(S)$  is an irreflexive ordering of  $LabelerIndex$ , and  $messages(S)$  is a sequence of *message\_type* indexed by  $LabelerIndex$ .

**global coherence:** For every  $SCAN$  event  $S$  that begins after  $m$  and indexes  $i \neq j$ , if  $L = \Omega_i(S)$  and  $M = \Omega_j(S)$  then:

$$i \triangleleft(S) j \text{ iff } L \Rightarrow M.$$

Moreover,  $message(\Omega_i(S)) = messages(S)(i)$  for every  $LabelerIndex$   $i$ .

**Definition 9.3** [*Stable cts*] An interpretation  $H$  of the  $L_{TS}$  language is said to stabilize to the cts properties if  $P_{cts}(m)$  holds for some moment  $m$  in  $H$ .

## 10 The self-stabilizing timestamp protocol

The *SCAN/LABEL* protocol described here in Figures 10 and 11 implements a self-stabilizing timestamp system. The protocol refers to  $N$  serial labeling processes, *Labeler<sub>i</sub>*, for  $1 \leq i \leq N$ , and several (serial) scanning processes that operate concurrently and are collectively called *Scanner*. We shall define the data types and specify the registers used, then explain informally the protocol, and finally prove its correctness and self-stabilization.

### 10.1 Data-types and registers used in the protocol

**message\_type** = Set of all possible data values that are written and read.

**index\_order** = Set of all linear orderings of  $\{1, \dots, N\}$ . ( $N$  is the number of *Labeler* processes.) Variable  $\triangleleft$  in the *SCAN* protocol is of this type.

**diagonal\_number** =  $\{1, \dots, 4N - 2\}$ . That is, a **diagonal\_number** is an integer in the range 1 to  $4N - 2$ .

**diag\_order** = Set of all linear orderings of  $\{1, \dots, 4N - 2\}$ . If  $D$  is of type **diag\_order**, then  $D$  is implemented by an array which is a permutation of  $\{1, \dots, 4N - 2\}$ . So,  $D[i]$  gives the place of the **diagonal\_number**  $i$  in this order, and hence  $i D j$  means  $D[i] < D[j]$ . So  $i$  is the last diagonal number in  $D$  if  $D[i] = 4N - 2$  etc.

**base\_type** = array  $\{1, \dots, N\}$  of **diag\_order**. That is, if  $b$  is of type **base\_type**, then  $b = (b[1], \dots, b[N])$  where each  $b[i]$  is a **diag\_order**.

**report\_type<sub>i</sub>** = array  $\{1, \dots, i\}$  of **diagonal\_number**, where  $1 \leq i \leq N$ . If  $r$  is in **report\_type<sub>i</sub>**, then  $r[k]$  is a **diagonal\_number**, for  $1 \leq k \leq i$ . We say that  $i$  is the length of  $r$ . For example, variable *report\_val* in *LABEL* is in **report\_type<sub>i</sub>**.

**report\_type** =  $\bigcup_{1 \leq i \leq N} \text{report\_type}_i$ .

**head\_type<sub>i</sub>** = record

trace : **diag\_order**;  
report : **report\_type<sub>i</sub>**;  
message : **message\_type**;

end.

So if  $h$  is a value in **head\_type<sub>i</sub>** then  $h$  is a record with three fields:  $h.trace$  is its **diag\_order**,  $h.report$  is its **report\_type<sub>i</sub>**, and  $h.message$  is its message (in **message\_type**). The diagonal number  $h.report[i]$  is called “the diagonal number of  $h$ ” and is denoted  $h.diagonal$ . There is an additional “consistency” requirement imposed on the **head\_type<sub>i</sub>** data type: we require that  $h.trace[h.diagonal] = 4N - 2$ . That is, the diagonal number of  $h$  is the last diagonal number in  $h.trace$ .

The function

*PUT\_LAST*( $d$  : **diagonal\_number**;  $D$  : **diag\_order**)

is called in line 2(c) of the *LABEL* protocol. This function changes  $D$  (of type **diag\_order**) by making  $d$  the last member of the ordering. Formally:

*PUT\_LAST* ( $d$ : **diagonal\_number**;  $D$  : **diag\_order**)

1. forall  $i$  ( $1 \leq i \leq 4N - 2$ ) do  
    if  $D[d] < D[i]$  then  $D[i] := D[i] - 1$ ;
2.  $D[d] := 4N - 2$ ;
3. return  $D$ .

It follows that if  $D_2 = \text{PUT\_LAST}(d, D_1)$ , then the only difference between the ordering of  $D_1$  and that of  $D_2$  is when  $d$   $D_1$   $x$  is transformed into  $x$   $D_2$   $d$ . That is:

**Lemma 10.1** *If  $D_2 = \text{PUT\_LAST}(d, D_1)$  then for every diagonal numbers  $x \neq y$  we have  $x$   $D_2$   $y$  iff  $(x$   $D_1$   $y \wedge x \neq d) \vee y = d$ .*

Observe how variable *diagonal* is put last in line 2(c) and is also assigned as  $head.report[i]$  in line 2(d) of the *LABEL* protocol. Hence  $head.report$  and  $head.trace$  satisfy the above consistency requirement about the place of the diagonal number of  $head$ .

Given any ordered set  $(A, <)$  a lexicographical ordering is defined on the finite sequences from  $A$ , as is well known. In fact, if  $<_1, \dots, <_N$  is any

sequence of orderings of the set  $A$ , then a lexicographical ordering can be defined on the finite sequences by using  $\prec_i$  to compare the  $i$ -th entries. In our case this yields the following definition.

**Definition 10.2 (LEX ordering)** *If  $b = (\prec_1, \dots, \prec_N)$  is of type **base\_type** (so each  $\prec_i$  is a **diag\_order**), then  $LEX(b)$  (interchangeably written as  $<_{LEX(b)}$ ) is an ordering defined on **report\_type** (sequences of lengths  $\leq N$  of **diagonal\_number**) by:*

$$s \text{ } LEX(b) \text{ } t$$

*iff*

*either for some  $1 \leq i$  in the domain of both sequences  $s[i] \neq t[i]$   
and for the least such  $i$*

$$s[i] \prec_i t[i],$$

*or else  $length(s) < length(t)$  and, for all  $i \leq length(s)$ ,  $s[i] = t[i]$*

Clearly for every  $b$  in **base\_type**,  $LEX(b)$  is a linear ordering of the set **report\_type**.

**Shared Variables:** For interprocess communication,  $Labeler_i$  employs both regular registers and  $UpDate/Scan$  operations that satisfy the *wss* properties. For self-stabilization, it suffices that the registers stabilize the regularity property (Section 6.1), and the  $UpDate/Scan$  operations stabilize and finally satisfy the weak snapshot axioms (*wss* of Section 6.2).

1.  $Labeler_i$  owns the following registers:  $LL(i, k)$  is a regular register for every  $1 \leq k < i$ . It is read by  $Labeler_k$ .  $Labeler_i$  writes on  $LL(i, k)$  values that are sets of  $\leq 3$  diagonal numbers.
2.  $Head(i)$  is a shared memory register that carries values of type **head\_type**. Process  $Labeler_i$  executes  $UpDate$  operations on  $Head(i)$ , and any process can execute a  $Scan$  operation in order to receive the values of all  $Head(i)$  registers for  $1 \leq i \leq N$ . In this context,  $Labeler_i$  is both an  $UpDater_i$  process (in its executions of  $UpDate$  operations on  $Head(i)$ ) and a  $Scanner$  process executing (selective) scan operations. We assume that these operations satisfy the weak snapshot intermediate axioms of Section 3, or at least that they stabilize to these axioms. That is (see Section 6.1), there is always a moment  $m$  such that  $P_{WSS-axioms}(m)$  holds.

*SCAN*

1.  $(h[1], \dots, h[N]) := \text{Scan}(\text{Head}(1), \dots, \text{Head}(N));$   
Let *basis* be defined by

$$\text{basis}[i] := h[i].\text{trace}$$

for every  $1 \leq i \leq N$ ;

2. Let  $\triangleleft$  be the order on  $\{1, \dots, N\}$  defined by:  
 $i \triangleleft j$  iff  $(h[i].\text{report} \text{ LEX}(\text{basis}) h[j].\text{report})$ ;
3. Let *messages* be the sequence of messages defined by:  
 $\text{messages}[i] = h[i].\text{message}$ ;
4. return  $\langle \triangleleft, \text{messages} \rangle$ .

Figure 10: The *SCAN* protocol. The local variables are as follows.  $h[i]$  is of type **head\_type<sub>i</sub>**. *basis* is of type **base\_type**.  $\triangleleft$  is of type **index\_order**.

Local variables for *Labeler<sub>i</sub>*: *head* is of type **head\_type<sub>i</sub>**.  $a_k, b_k, c_k, h[k]$  are of type **head\_type<sub>k</sub>**. *report\_val[k]*, and *diagonal* are **diagonal\_number**.  $\ell[j]$  is a set of  $\leq 3$  **diagonal\_number**. *evade* is a set of  $< 4N - 2$  **diagonal\_number**.

The *Scanners* are silent, they have no registers to write on.

## 10.2 The protocol reviewed

The protocol is the text of the *SCAN* and *LABEL* operations in Figures 10 and 11. We assume that there are  $M$  serial processes that only execute *SCAN* operations, and  $N$  serial processes that only execute *LABEL* operations. This assumption is convenient, but we can certainly combine *SCAN* and *LABEL* processes into one “super-process” that executes both operations. Each process has its own set of local variables.

The processes have two types of devices for interprocess communication: regular registers on which they execute read and write actions, and shared

```

LABEL ( $p$  : message_type)    (by Labeleri);

1. concurrently do {down, up}

  down :
    (a)  $(a_1, \dots, a_{i-1}) := \text{Scan}(\text{Head}(1), \dots, \text{Head}(i-1));$ 
    (b) forall  $1 \leq k < i$  do
        write  $\text{LL}(i, k) := \{\text{report\_val}[k], a_k.\text{diagonal}\};$ 
    (c)  $(b_1, \dots, b_{i-1}) := \text{Scan}(\text{Head}(1), \dots, \text{Head}(i-1));$ 
    (d) forall  $1 \leq k < i$  do
        if  $a_k.\text{diagonal} \neq b_k.\text{diagonal}$ 
        then write  $\text{LL}(i, k) := \{\text{report\_val}[k], a_k.\text{diagonal}, b_k.\text{diagonal}\};$ 
    (e)  $(c_1, \dots, c_{i-1}) := \text{Scan}(\text{Head}(1), \dots, \text{Head}(i-1));$ 
    (f) forall  $1 \leq k < i$  do
        i. if  $a_k.\text{diagonal} = b_k.\text{diagonal}$  or  $b_k.\text{diagonal} = c_k.\text{diagonal}$ 
            then  $\text{report\_val}[k] := b_k.\text{diagonal};$ 
        ii. if  $a_k.\text{diagonal} \neq b_k.\text{diagonal}$  and  $b_k.\text{diagonal} \neq c_k.\text{diagonal}$ 
            then  $\text{report\_val}[k] := a_k.\text{diagonal};$ 

  up :
    (a) forall  $(i < k \leq N)$  do read  $\ell[k] := \text{LL}(k, i);$ 
    (b)  $(h[i+1], \dots, h[N]) := \text{Scan}(\text{Head}(i+1), \dots, \text{Head}(N));$ 

2. (* Calculation of head *)
    (a)  $\text{evade} := \{\text{diagonal}\} \cup \bigcup \{\ell[j] \mid i < j \leq N\} \cup$   

         $\{h[k].\text{report}[i] \mid i < k \leq N\};$ 
    (b) Assign to diagonal some diagonal_number not in evade.
    (c)  $\text{head.trace} := \text{PUT\_LAST}(\text{diagonal}; \text{head.trace});$ 
    (d)  $\text{report\_val}[i] := \text{diagonal}; \text{head.report} := \text{report\_val};$ 
    (e)  $\text{head.message} := p;$ 

3. UpDate(Head( $i$ ); head).

```

Figure 11: The *LABEL* protocol for *Labeler<sub>i</sub>* where  $1 \leq i \leq N$ .

variables  $Head(i)$  on which they execute  $UpDate$  and  $Scan$  actions. We assume that these interprocess communication actions stabilize in a normal execution of the protocol, and we will prove that the  $SCAN$  and  $LABEL$  operation executions stabilize to the concurrent timestamp properties.

Recall that we make a distinction between  $SCAN$  and  $Scan$  operations. A  $SCAN$  is an execution of the protocol of Figure 10, and a  $Scan$  is an assumed  $wss$  action (corresponding to an instruction of the protocol executed). We could assume that these  $Scan/UpDate$  operations are implemented with the protocols of Section 7, but in fact we only need to know that they self-stabilize to the  $wss$  intermediate axioms, that is, for some moment  $m$ ,  $P_{wss-axioms}(m)$  holds (see Section 6.3).

The  $SCAN$  operation consists of a single  $Scan$  action which returns the values of all  $N$  registers  $Head(1), \dots, Head(N)$  (see line 1 of the  $SCAN$  protocol). These values are assigned to local variables  $h[1], \dots, h[N]$ . So  $h[i]$  is the value obtained from register  $Head(i)$  and is of type **head\_type<sub>i</sub>**. The protocol assigns to variable  $basis$  the array of the trace fields obtained. So  $basis[i]$  is a linear order on  $\{1, \dots, 4N - 2\}$ , and  $LEX(basis)$  is a linear ordering, as defined in 10.2, of the set **report\_type** (sequences of length  $\leq N$  of diagonal numbers). The  $SCAN$  protocol uses  $LEX(basis)$  to define a linear ordering  $\triangleleft$  on the set of indexes  $\{1, \dots, N\}$ . A  $SCAN$  operation execution returns in its variable  $messages$  the sequence  $\langle h[1].message, \dots, h[N].message \rangle$ , and it decides on the ordering  $i \triangleleft j$  by comparing  $s_i = h[i].report$  (a sequence of length  $i$  of diagonal numbers) with  $s_j = h[j].report$  (a sequence of length  $j$ ): if there is an index  $k \leq i, j$  such that  $s_i[k] \neq s_j[k]$ , and  $k$  is the least such index, then  $i \triangleleft j$  iff  $h[k].trace$  puts  $s_i[k]$  before  $s_j[k]$ ; otherwise we define  $i \triangleleft j$  iff  $s_i$  is a proper initial segment of  $s_j$ .

If  $S$  is an execution of the  $SCAN$  protocol and  $S$  returns the pair  $\langle \triangleleft_0, messages_0 \rangle$ , then we shall define  $\triangleleft(S) = \triangleleft_0$  and  $messages(S) = messages_0$ . These two functions  $\triangleleft$  and  $messages$  will be shown to satisfy the required concurrent timestamping properties of Section 9.2.1.

The  $LABEL$  protocol is more complex than the  $SCAN$ . It takes two parameters: the message  $p$  to be labelled and the index  $1 \leq i \leq N$  of the executing process  $Labeler_i$ . Line 1 of the  $LABEL$  protocol (Figure 11) consists of two instructions, **down** and **up**, which can be executed concurrently or in any arbitrary interleaving of their inner actions. Three  $Scan$  actions are executed in lines **down** (a), (c), (e), and an additional  $Scan$  is executed in line **up**(b). These are selective  $Scan$  operations as defined in section 8.1. That is, the  $Scan$  actions in the **down** instruction, read only the registers  $Head(k)$

for smaller indexes  $k < i$ . The *Scan* operation in the **up** part read only the registers of higher indexes. So *Labeler*<sub>1</sub> has a degenerate (null) **down** part, and *Labeler*<sub>N</sub> a degenerate **up** part.

Observe that in lines **down**(b) and (d) we have two sets of writes on the regular registers  $LL(i, k)$  directed to *Labeler*<sub>k</sub> with  $k < i$ . The value of each such write is a set of at most three numbers. (The writes in **down**(d) are executed only on condition that the required inequalities hold.)

In line **down**(b) we see that a set of  $\leq 2$  diagonal numbers is written on  $LL(i, k)$  for  $1 \leq k < i$  (in any order). Here, *report\_val*[k] is a local variable whose value is determined in the previous operation (or is the initial value for the first operation execution), and  $a_k$ .diagonal is, by definition,  $a_k$ .report[k].

In the **up** part, *Labeler*<sub>i</sub> reads (in any order) all  $LL(k, i)$  registers for  $k > i$ , and then scans the *Head*(k) registers for  $k > i$  (in a single selective action).

In line 2(a) the diagonal numbers collected in the **up** part together with *diagonal* are assigned to *evade*. It follows that *evade* contains at most  $1 + 3(N - i) + (N - i) \leq 4N - 3$  numbers. Since there are  $4N - 2$  diagonal numbers, it is always possible to choose one which is not in *evade*. This is how the new value of *diagonal* is chosen in line 2(b).

Observe that any two successive *LABEL* operations have distinct *diagonal* values, since *evade* contains the diagonal value of the previous operation.

Line 2(c) invokes procedure *PUT\_LAST* which was explained above. The value of variable *head* is written on register *Head*(i) in the *UpDate* operation in line 3.

### 10.3 Correctness of the protocol

**Theorem 10.3** *The SCAN and LABEL protocols presented here implement a self-stabilizing timestamp. That is, if H is any normal execution of the protocols then for some moment m  $P_{cts}(m)$  holds.*

The proof of the theorem is extended in this section. Assume throughout this section that *H* is a normal system execution of the protocol. Normal executions were defined in general terms in Definition 6.2, and we shall repeat this definition here but now with our specific application in mind. There is a moment  $m_0$  such that the following hold in *H*.

1. All communication devices in *H* behave correctly after  $m_0$ . Specifically:

- (a) The *Scan* and *UpDate* actions on the set of  $Head(i)$  registers satisfy  $P_{WSS-axioms}(m_0)$ . (See Section 6.3.)
  - (b)  $P_R(m_0)$  holds for each of the regular registers  $R = LL(p, q)$ . (See Section 6.1.)
2. Every crashed process contains only events that have ended before  $m_0$ . (If *Scanner* is crashed, then the correctness condition  $P_{cts}(m_0)$  applies trivially, and we can exclude this possibility.)
  3. After moment  $m_0$ , the events of any non-crashed process *Labeler<sub>i</sub>* and *Scanner* represent executions of operations *LABEL* and *SCAN* that are done in accordance with the protocols of Figures 11 and 10.
  4. The high-level events of  $H$  consist of the following events. For a non-crashed *Labeler<sub>i</sub>*, the events are those high-level events that represent justified executions of a *LABEL* operation, in accordance with its protocol. And for a crashed *Labeler<sub>i</sub>* there is a single high-level event, called the ultimate event of *Labeler<sub>i</sub>*. It consists of all ultimate *write* and **update** events on the registers  $LL(i, k)$  and  $Head(i)$  of *Labeler<sub>i</sub>*.  
The *SCAN* events are the high-level, justified executions of the corresponding protocol.

Let  $m'_0 > m_0$  be some moment so that every non-crashed *Labeler<sub>i</sub>* contains an operation execution that begins after  $m_0$  and ends before  $m'_0$ . The last such *LABEL* operation execution is denoted  $X^0(\text{Labeler}_i)$ .

For a crashed *Labeler<sub>i</sub>* we define an “ultimate” higher-level event denoted  $X^0(\text{Labeler}_i)$ . This event consists of the following:

1. The ultimate *UpDate* event on  $Head(i)$ .
2. For every *Scanner* index  $k < i$ , the ultimate write event on  $LL(i, k)$ .

In this section, the term “operation execution” refers to an operation execution that begins after  $m_0$ , and the term “higher-level event” refers either to an operation execution or to an ultimate event of a crashed process. The term “normal operation execution” (or “normal event”) refers to one that begins after  $m'_0$ .

We shall define the following functions on higher-level events in  $H$ :

1. Atemporal valued functions: *diagonal*, *report\_val*, *message*, *trace*, *messages*, and  $\triangleleft$ .
2. Event valued functions are defined on the normal operation executions:  $\Omega_i$  (for every index  $i$ ), *E\_report*, and *E\_avoid*.

**Definition 10.4** For any high-level *Labeler<sub>i</sub>* event  $L$  (a *LABEL* operation execution or the ultimate event of a crashed *Labeler*) let  $u$  be the *UpDate* action in  $L$  on register  $Head(i)$ . So  $u$  is the last action corresponding to line 3 of the *LABEL* protocol if  $L$  is an operation execution, and  $u$  is the last *UpDate* of *Labeler<sub>i</sub>* if it is crashed). We define the following values of functions on  $L$ :

1. *diagonal*( $L$ ) is the diagonal field value written by  $u$  on  $Head(i)$ ,
2. *report\_val*( $L$ ) is the report field value written by  $u$  on  $Head(i)$ ,
3. *message*( $L$ ) is the message field written by  $u$  on  $Head(i)$ .
4. *trace*( $L$ ) is the trace field value written by  $u$  on  $Head(i)$ .

For every high-level  $L$  in *Labeler<sub>i</sub>*,  $diagonal(L) = report\_val(L)(i)$  is the last diagonal number in  $trace(L)$ . This is a requirement made on the type of values **head\_type**, and we assume that even crashed values are in their types.

Functions with names that appear in the protocol itself either as variables or as fields, are also defined by the following convention: For a variable (or field)  $v$  and an operation execution  $X$ , the value of  $v$  at the end of  $X$  is denoted  $v(X)$  or  $v^X$ . This takes care of *diagonal*, *report\_val*, *message*, and *trace* for *Labeler* operations, and  $\triangleleft$ , and *messages* for *SCAN* operation executions.

If  $L$  is a *LABEL* operation execution then we have two definitions for these functions. One is given above in 10.4 and relies on the value written on  $Head(i)$ , and the second depends on the value of the relevant variable at the end of the execution. For a *LABEL* operation execution these two definitions coincide. For example,  $trace(L)$  can be defined as either the value of the trace field value of the *UpDate* operation in  $L$ , or as the final value of variable field *head.trace* in  $L$ . If, however,  $L$  is the ultimate event of crashed *Labeler<sub>i</sub>*, then there may be a discrepancy between the value of the *UpDate*

and the value of the variable, and what counts is the value written by the final *UpDate* operation.

Now if  $S$  is a *SCAN* operation execution we define

$$\triangleleft(S) = \triangleleft^S.$$

We also define

$$messages(S) = messages^S.$$

$\triangleleft(S)$  and  $messages(S)$  are the ordering and sequence of messages returned by  $S$ .

We now define the event-valued functions on the normal operation executions. That is, the functions  $\Omega_i$ ,  $E\_avoid$ , and  $E\_report$  (the letter  $E$  indicates that these functions return high level events).

The “return” functions  $\Omega_i$  are defined on any normal *SCAN* operation execution  $S$  as follows, for any index  $1 \leq i \leq N$ . Let  $r$  be the *Scan* action in  $S$  executed for line 1. Then  $\Omega(r, i)$  is an *UpDate* event executed by  $Labeler_i$  on  $Head(i)$  (by item  $2_m$  in the stabilizing *wss* intermediate axioms list of Figure 5). So there is a higher-level event denoted  $[\Omega(r, i)]$  that contains  $\Omega(r, i)$ , and we define

$$\Omega_i(S) = [\Omega(r, i)].$$

In details the definition and accompanying arguments are as follows. Suppose first that  $Labeler_i$  is non-crashed. As  $X = X^0(Labeler_i)$  begins after  $m_0$  and ends before  $m'_0$ ,  $X$  is a justified operation execution and  $X \rightarrow S$  since  $S$  begins after  $m'_0$ . Now,  $X$  contains an *UpDater<sub>i</sub>* event  $U$  (executing line 3), and since  $U \rightarrow r$ ,  $\Omega(r, i) \rightarrow U$  is impossible by property 2(c) of the *wss* intermediate axioms. So  $\Omega(r, i)$  is either in  $X$  or in a later operation execution, and in any case  $\Omega_i(S)$  is a  $Labeler_i$  operation execution. Next suppose that  $Labeler_i$  is crashed. Then  $\Omega(r, i)$  is the ultimate *UpDate* operation on  $Head(i)$ , and hence  $\Omega_i(S)$  is the ultimate  $Labeler_i$  event  $X^0(Labeler_i)$ .

For any normal *LABEL* operation execution  $L$  by  $Labeler_i$  we define the events  $E\_avoid(L, j)$  for  $i < j$ , and  $E\_report(L, k)$  for  $k \leq i$  as follows.

If  $j > i$ , then  $E\_avoid(L, j)$  is defined as follows. Let  $r$  be the read of register  $LL(j, i)$  done in line **up** in  $L$ . Then  $E\_avoid(L, j) = [\omega(r)]$ . That is,  $\omega(r)$  is in some  $Labeler_j$  higher-level event  $V$  and we set  $E\_avoid(L, j) = V$ .

Next,  $E\_report(L, k)$  is defined for every index  $k \leq i$ . For  $k = i$ , put  $L = E\_report(L, i)$ . That is,  $L$  by  $Labeler_i$  reports itself. Now assume that  $k < i$ . Let  $r_1, r_2$ , and  $r_3$  be the three scans in  $L$ , executed in lines **1.down(a)**,

**1.down**(c), and **1.down**(e) respectively. Let  $a_k$ ,  $b_k$ , and  $c_k$  be the values of  $Head(k)$  obtained in these scan operations. Define  $d_1 = a_k.diagonal$ ,  $d_2 = b_k.diagonal$ , and  $d_3 = c_k.diagonal$ .

**Case 1** :  $d_1 = d_2$ . Define in this case

$$E\_report(L, k) = [\Omega(r_2, k)].$$

Assume now that  $d_1 \neq d_2$ .

**Case 2** :  $d_1 \neq d_2$ , but  $d_2 = d_3$  or  $d_1 = d_3$ . Define

$$E\_report(L, k) = [\Omega(r_3, k)].$$

**Case 3** :  $d_1, d_2, d_3$  are three different values. Define  $E\_report(L, k)$  to be the  $\rightarrow$  rightmost  $Labeler_k$  operation execution  $C$  such that  $diagonal(C) = d_1$  and  $end(C) \leq end(r_3)$ . Clearly  $[\Omega(r_1, k)] \rightarrow= C$ . Why? Because  $W = \Omega(r_1, k)$  is an  $UpDate$  action that satisfies  $diagonal(W) = d_1$  and  $end(W) < end(r_1)$  by intermediate axiom  $2_m(b)$  in the list of Figure 6.3.

This ends the definition of the higher-level functions, and now we investigate some of their properties. This will lead to a proof that  $H$  is a stabilizing *cts* execution.

**Lemma 10.5** *Let  $L$  be a normal LABEL operation execution by  $Labeler_i$ ,  $k < i$ , and  $V = E\_report(L, k)$ . Then  $report\_val(L)(k) = diagonal(V)$ .*

**Proof:** Following the definition of  $V = E\_report(L, k)$  and the notation used there, we obtain in accordance with the three cases in that definition that

$$diagonal(V) = \begin{cases} d_2 & d_1 = d_2 \\ d_3 & d_1 \neq d_2 \text{ and } d_3 \in \{d_1, d_2\} \\ d_1 & |\{d_1, d_2, d_3\}| = 3 \end{cases}$$

Now consider instruction **1(f)** executed by  $L$  in determining  $report\_val[k]$  for  $1 \leq k < i$ . If  $d_1 = d_2$  or  $d_2 = d_3$  then  $report\_val[k] = d_2 = diagonal(V)$ . Assume next that  $d_1 \neq d_2$  and  $d_2 \neq d_3$ . Then **1down**(f)(ii) applies and  $report\_val[k] = d_1$ . If  $d_3 = d_1$ , then  $diagonal(V) = d_3 = d_1$  as required. But if  $d_3 \neq d_1$ , then  $diagonal(V) = d_1 = report\_val[k]$  again. ■

Our next lemma follows directly from the definitions of the functions *trace* and *diagonal* on the LABEL operation executions.

**Lemma 10.6 (the Trace Lemma)** *Let  $V$  and  $V^+$  be a LABEL operation execution and its successor in  $Labeler_i$ . Then*

$$trace(V^+) = PUT\_LAST(diagonal(V^+), trace(V)).$$

**Proof:** By definition,  $trace(V^+)$  is the value of variable  $head.trace$  as it is determined by  $V^+$  in executing line 2(c). As parameters to  $PUT\_LAST$ ,  $V^+$  supplies  $diagonal^{V^+}$  and  $head.trace^V$ . So the lemma is evident. ■

The following is a direct consequence of Lemma 10.1.

**Lemma 10.7** *Let  $V$  and  $V^+$  be a LABEL operation execution and its successor in  $Labeler_k$ . If  $d_1, d_2$  are diagonal numbers such that  $d_1 trace(V) d_2$  but  $d_2 trace(V^+) d_1$ , then  $d_1 = diagonal(V^+)$ .*

**Lemma 10.8 (the Trace Constancy Lemma)** *Suppose that  $V_1, V_2, V^*$  are  $Labeler_k$  LABEL operation executions such that*

$$V_1 \rightarrow V_2 \rightarrow = V^*$$

and

$$diagonal(V_2) trace(V^*) diagonal(V_1). \quad (5)$$

Then for some  $Labeler_k$  operation execution  $V_3$ ,

$$V_2 \rightarrow V_3 \rightarrow = V^*$$

and  $diagonal(V_1) = diagonal(V_3)$ .

**Proof:** Say  $d_1 = diagonal(V_1)$  and  $d_2 = diagonal(V_2)$ . Suppose that  $V_2, V_3, \dots, V_\ell = V^*$  is the sequence of LABEL operation executions in  $Labeler_k$  leading from  $V_2$  to  $V^*$ , where  $V_{i+1}$  is the successor of  $V_i$ . Let  $2 \leq \ell_1 \leq \ell$  be the first index for which  $d_2 trace(V_{\ell_1}) d_1$  holds. (By (5)  $\ell_1$  exists.)

Consider line 2(c) of the LABEL protocol. It follows immediately for every LABEL operation execution  $L$  that  $diagonal(L)$  is the greatest diagonal number in the  $trace(L)$  ordering. Since,  $trace(L)$  is always irreflexive, (5) implies that  $d_1 \neq d_2$ , and hence  $d_1 trace(V_2) d_2$ . Thus  $\ell_1 > 2$ , and by the minimality of  $\ell_1$ ,  $\ell_0 = \ell_1 - 1$  satisfies  $d_1 trace(V_{\ell_0}) d_2$ . So the previous lemma implies that  $d_1 = diagonal(V_{\ell_1})$  and hence  $V_{\ell_1}$  is as required. ■

We note for future reference the following

**Lemma 10.9** *For every normal SCAN operation execution  $S$ ,  $end(\Omega_k(S)) < end(S)$ .*

**Proof:** Let  $r$  be the *Scan* event in  $S$ , and let  $u = \Omega(r, k)$  be the corresponding *UpDate* event by  $Labeler_k$ . The stabilizing weak snapshot intermediate axiom  $2_m(b)$  of Figure 5 implies that  $end(u) < end(r)$ . There are two possibilities for  $\Omega_k(S) = [u]$ . The first is when  $Labeler_k$  is crashed and  $[u]$  is the ultimate  $Labeler_k$  event. In this case  $end[u] < m_0 < begin(S)$  and the conclusion of the lemma is obvious. Now if  $Labeler_k$  is not crashed, then  $X = X^0(Labeler_k)$  satisfies  $end(X) < m'_0$ . Yet  $m'_0 < begin(r)$ , and hence  $X \rightarrow r$  follows. Let  $w$  be the *UpDate* action in  $X$  (corresponding to line 3). Since  $w \rightarrow r$ ,  $w \rightarrow u$  follows axiom  $2_{m_0}(c)$ . So  $u$  belongs to some *LABEL* operation execution and it is the last event there. This implies  $end(u) < end(r)$  by  $2_{m_0}(c)$ , and hence the lemma follows. ■

**Lemma 10.10** *Let  $W$  be a normal  $Labeler_i$  operation execution, and let  $w$  be the *UpDate* action of  $W$  (its last event). For every index  $k < i$ , if  $V = E\_report(W, k)$  then  $V$  is a  $Labeler_k$  event such that  $end(V) < end(W)$ , and in fact  $end(V) < begin(w)$ .*

*For a fixed  $k$  the function taking  $X$  to  $E\_report(X, k)$  is regular from the normal *LABEL* events  $X$  in  $Labeler_i$  into the high-level events of  $Labeler_k$ . (This means that  $X \rightarrow E\_report(X, k)$  never holds, and there is no  $Labeler_k$  event  $L$  such that  $E\_report(X, k) \rightarrow L \rightarrow X$ .)*

**Proof:** Suppose first that  $Labeler_k$  is crashed. Then, if  $r$  is any *Scan* action in  $W$ , we get  $\Omega(r, k) \in X^0(Labeler_k)$ . Thus, in the definition of  $E\_report(W, k)$  we get that Case 1 holds, namely  $d_1 = d_2$ , and  $E\_report(W, k)$  is the ultimate event  $X^0(Labeler_k)$ . The lemma is trivially true in this case. Assume next that  $Labeler_k$  is not crashed.

The definition of  $V = E\_report(W, k)$  is in three cases, and we check in each case that  $end(V) < begin(w)$  and that there is no  $Labeler_k$  operation execution  $T$  such that  $V \rightarrow T \rightarrow W$ . In Case 1 and Case 2 there is some *Scan*  $r$  in  $W$  such that  $V = [\Omega(r, k)]$ . That is,  $\Omega(r, k) \in V$  is the last action there. Certainly, there is no  $T$  with  $V \rightarrow T \rightarrow W$ , by intermediate axiom  $2_{m_0}(c)$ . In Case 3,  $E\_report(W, k)$  is defined to be the last operation execution  $C$  such that  $diagonal(C) = d_1$  and  $end(C) \leq end(r_3)$ . So clearly,  $end(C) < begin(w)$ . But there cannot be any  $Labeler_k$  event  $T$  such that  $C \rightarrow T \rightarrow W$  because  $\Omega(r_1, k) \rightarrow C$  (see Case 3 in the definition). ■

**Lemma 10.11** *Let  $S$  be a normal SCAN event. Suppose that  $L = \Omega_i(S)$  is a normal LABEL operation execution. Then, for every index  $k \leq i$ , if  $V = E\_report(L, k)$  then  $V \rightarrow \Omega_k(S)$ .*

**Proof:** For  $k = i$ ,  $V = L$  and the lemma is obvious in this case. So we may assume  $k < i$ . Let  $w$  be the *UpDate* event in  $L$ . Lemma 10.10 implies that  $end(V) < begin(w)$ . Let  $r$  be the *Scan* action in  $S$ . Then  $w = \Omega(r, i)$  by the definition of  $\Omega_i$ . So  $begin(w) < begin(r)$  by  $2_m(b)$ , and hence  $V \rightarrow r$ . This implies that  $V \rightarrow \Omega_k(S)$ . ■

**Lemma 10.12** *Let  $W$  be a normal LABEL operation execution by Labeler $_i$  and let  $k < i$  be a lower index. Suppose that  $V = E\_report(W, k)$  and  $d = diagonal(V)$ . Let  $d_1, d_2, d_3$  be the diagonal numbers obtained as in the definition of  $V = E\_report(W, k)$  (that is, the diagonal fields of  $Head(k)$  as obtained in the three scans of  $W$  in line **down**). Let  $W^+$  be an operation execution that is the successor of  $W$  in Labeler $_i$ . Then  $d$  is present in the second write in  $W$  on  $LL(i, k)$  and in the two writes on  $LL(i, k)$  that are in  $W^+$ .*

*In case  $d_1 = d_2$ , and in case ( $d_1 \neq d_2$  and  $d_2 \neq d_3$ ), we have that  $d$  is also present in the first write on  $LL(i, k)$  in  $W$ .*

**Proof:** The first write on  $LL(i, k)$ , corresponding to line **down(b)**, contains  $\leq 2$  diagonal numbers, and the second write contains  $\leq 3$  numbers. We say that  $d$  is present in a write if it is one of those values.

By Lemma 10.5,  $d = report\_val(W)(k)$ . This diagonal number is determined in line **down(f)** in  $W$ , and it is one of  $d_2$  and  $d_1$ . Since both of these values are present in the second write in  $W$ ,  $d$  is present in the second write in  $W$ . By means of variable  $report\_val[k]$ ,  $d$  is present in the two writes on  $LL(i, k)$  in  $W^+$ .

In case  $d_1 = d_2$  and in case ( $d_1 \neq d_2 \wedge d_2 \neq d_3$ ) we have  $d = d_1$ . So in this case  $d$  is present in the first write on  $LL(i, k)$  in  $W$  as well. ■

**Lemma 10.13 (The Report Lemma)** *Let  $W_1$  be a normal LABEL event by Labeler $_i$ , and  $k < i$  be a lower index. Suppose that*

1.  $V_1 = E\_report(W_1, k)$ ,
2.  $V_1 \rightarrow V_2$ ,  $V_2$  in Labeler $_k$  is a LABEL operation execution,
3.  $diagonal(V_1) = diagonal(V_2)$ , and

4.  $E\_avoid(V_2, i) = W_2$ .

Then  $W_1^+$  exists and  $W_1^+ \rightarrow W_2$ .

**Proof:** Consider the definition of  $V_1 = E\_report(W_1, k)$ , and let  $r_1, r_2, r_3$  be the three scans and  $d_1, d_2, d_3$  the three diagonals be as in that definition. Assume first that  $d_1 = d_2$ , or  $d_2 = d_3$ , or  $d_1 = d_3$ . These three cases are similar to each other, and we suppose, for example, that  $d_1 = d_2$ . Then  $V_1 = [\Omega(r_2, k)]$  and  $diagonal(V_1) = d_2$  follows. Let  $w_1$  be the write onto  $LL(i, k)$  in  $W_1$  corresponding to line **down**(b). So  $r_1 \rightarrow w_1 \rightarrow r_2$ . Then  $d_1 = d_2$  is present in  $w_1$  (it is one of the diagonal values of  $w_1$ ). Let  $w_2$  be the second write in  $W_1$  on  $LL(i, k)$  done in line **down**(d). Then  $d_1$  is present in  $w_2$  as well. This excludes the possibility that  $W_1 = E\_avoid(V_2, i)$  (because in this case  $d_1 = d_2$  is “avoided” by  $V_2$ , and  $d_1 = diagonal(V_2)$  would be impossible). But in this case,  $W_1$  sets  $report\_val[k] := d_1$  and since this value is carried in the successor,  $W_1^+$ , of  $W_1$ ,  $W_1^+ = E\_avoid(V_2, i)$  is also impossible. So we must prove that  $W_1 \rightarrow W_2$  in order to conclude that  $W_1^+$  exists and  $W_1^+ \rightarrow E\_avoid(V_2, i)$ , as the lemma requires.

Since  $V_1$  and  $V_2$  have the same diagonal number,  $V_2$  is not the successor of  $V_1$  in  $Labeler_k$  and hence  $V_1^+ \rightarrow V_2$  (where  $V_1^+$  is the successor of  $V_1$ ). Now  $V_1^+ \rightarrow r_2$  would contradict  $V_1 = \Omega(r_2, k)$ . Hence  $begin(r_2) \leq end(V_1^+)$ , and thus  $w_1 \rightarrow V_2$  follows. This implies that  $W_1 \rightarrow W_2$ .

Assume next that  $d_1 \neq d_2$ ,  $d_2 \neq d_3$ , and  $d_1 \neq d_3$ . Then  $report\_val(W_1)(k) = d_1$ , and Case 3 in the definition of  $V_1 = E\_report(W_1, k)$  applies. Thus  $V_1$  is the  $\rightarrow$  rightmost  $Labeler_k$  operation execution that ends before  $r_3$  ends and has  $d_1$  as its diagonal. So  $diagonal(V_1) = d_1$ , and this diagonal is present in both write in  $W_1$  on  $LL(i, k)$ . As before, this excludes the possibility that  $E\_avoid(V_2, i) = W_1$  or  $E\_avoid(V_2, i) = W_1^+$ . So, again, we must prove that  $W_2 \rightarrow W_1$  is impossible. For a contradiction, assume that this is the case. Let  $s_2 \in V_2$  be the read of  $LL(i, k)$ . Then

$$begin(s_2) \leq end(w_1) \tag{6}$$

where  $w_1$  is (as above) the first write onto  $LL(i, k)$  in  $W_1$  (or else  $w_1 \rightarrow \omega(s_2)$  and  $W_1 \rightarrow W_2$  follows). Let  $u_2 \in V_2$  be the  $UpDate$  operation on  $Head(k)$ . It is the last event in  $V_2$ . So  $end(u_2) \leq end(r_3)$  is impossible (or else  $V_1$  would not be the rightmost  $Labeler_k$  operation with  $d_1 = diagonal(V_1)$  and  $end(V_1) \leq end(r_3)$ ). Hence

$$\text{end}(r_3) < \text{end}(u_2). \quad (7)$$

By the *UpDate/Scan* semantics, this excludes the possibility that  $\Omega(r_3, k) = u_2$ , as well as the possibility that  $\Omega(r_2, k) = u_2$ . Let  $V_2^-$  be the *Labeler<sub>k</sub>* operation execution that is the immediate predecessor of  $V_2$ . Let  $u_2^-$  be the *UpDate* operation in  $V_2^-$ . We claim that  $u_2^- = \Omega(r_2, k) = \Omega(r_3, k)$ . But this would contradict our assumption that  $d_2 \neq d_3$ .

To prove that claim, observe first that  $u_2^- \rightarrow s_2$  and (6) imply that  $u_2^- \rightarrow r_2 \rightarrow r_3$ . Hence  $\Omega(r_2, k)$  is  $u_2^-$  or a later *UpDate*. But  $\Omega(r_2, k)$  cannot be a later *UpDate* because  $\text{end}(r_2) < \text{end}(u_2)$  (by 7). Thus  $u_2 = \Omega(r_2, k)$ , and  $u_2 = \Omega(r_3, k)$  by a corresponding argument. ■

For every normal *LABEL* operation execution,  $L$ , by *Labeler<sub>i</sub>* we define

$$E\_reports(L) = (E\_report(L, 1), \dots, E\_report(L, i)),$$

a sequence of *LABEL* events with the  $j$ -th entry in *Labeler<sub>j</sub>*.

**Definition 10.14** *Using the fact that each *Labeler<sub>i</sub>* is serial, a global-order,  $\Rightarrow$ , is defined on the normal *LABEL* events in  $H$  as follows. Suppose that  $E\_reports(L) = (L_1, \dots, L_i)$ ,  $E\_reports(L') = (L'_1, \dots, L'_{i'})$ . Then  $L \Rightarrow L'$  if and only if*

- *either there is some  $k$ ,  $k \leq \min\{i, i'\}$ , such that  $L_k \neq L'_k$ , and for the first such  $k$   $L_k \rightarrow L'_k$ , or else*
- *for all  $k \leq \min\{i, i'\}$   $L_k = L'_k$  and  $i < i'$ .*

We show that  $\Rightarrow$  is a linear ordering of the normal *Labeler* operation executions, and that it extends  $\rightarrow$  on these events. The fact that  $\Rightarrow$  is a linear ordering is a standard result on the lexicographical order of finite sequences, and the fact that each *Labeler<sub>i</sub>* is sequential is used here.

To show that  $\Rightarrow$  extends  $\rightarrow$  on the normal *LABEL* events, we argue as follows.

**Lemma 10.15** *If  $L_1 \rightarrow L_2$  are normal *LABEL* events then  $L_1 \Rightarrow L_2$ .*

**Proof:** Put  $i = \text{index}(L_1)$ , and  $j = \text{index}(L_2)$ . Since each *Labeler<sub>k</sub>* is serial and  $E\_report(L, k)$  is a regular function of  $L$  such that  $E\_report(L, k)$  ends before  $L$  does (Lemma 10.10), for every  $k \leq m = \min\{i, j\}$  we have that

$$E\_report(L_1, k) \rightarrow E\_report(L_2, k).$$

If strong precedence (i.e.  $\rightarrow$ ) holds for some  $k \leq m$ , then  $L_1 \Rightarrow L_2$  as required. Strong precedence holds when  $i \geq j$ . For if  $i = j$ , then

$$L_1 = E\_report(L_1, i) \rightarrow L_2 = E\_report(L_2, j),$$

and if  $i > j$  then  $end(E\_report(L_1, j)) < end(L_1)$  and  $L_1 \rightarrow L_2$  imply that  $E\_report(L_1, j) \rightarrow L_2 = E\_report(L_2, j)$ .

But if  $i < j$  and strong precedence never holds, that is

$$E\_report(L_1, k) = E\_report(L_2, k)$$

for every  $k \leq i$ , then  $L_1 \Rightarrow L_2$  as well. ■

Recall that if  $Labeler_i$  is crashed then it contains a single higher-level event, namely its ultimate  $X^0(Labeler_i)$ . We have not yet defined the place of these crashed ultimate events in  $\Rightarrow$ , as this ordering was defined only on the normal  $Labeler$  events. Our next objective is to extend  $\Rightarrow$  over the ultimate events of crashed processes as well. While it is obvious that each ultimate event should be arranged behind every normal one, the ordering of the ultimate events between themselves needs a special attention and will be defined later.

**Definition 10.16** *Suppose that  $1 \leq i \leq N$  is a  $Labeler$  index and  $d$  is a **diagonal number**. We say that  $d$  is “unbounded” in  $Labeler_i$  if  $d = diagonal(L)$  for an infinite number of LABEL operation executions  $L$  that are in  $Labeler_i$ . Otherwise, if  $d$  is the diagonal number of only a finite number of  $Labeler_i$  events, then we say that  $d$  is bounded in  $Labeler_i$ . For every  $Labeler$  index  $i$  let  $Bd(i)$  be the set of those diagonal numbers that are bounded in  $Labeler_i$ . If  $Labeler_i$  is crashed, then every diagonal number is bounded in  $Labeler_i$ , and so  $Bd(i) = \mathbf{diagonal\ number}$  in this case.*

Let  $m_1 > m'_0$  be a moment such that for every  $Labeler$  index  $i$ , if  $L$  is any  $Labeler_i$  operation execution that starts after  $m_1$ , then  $diagonal(L) \notin Bd(i)$ .

The following lemma implies that if  $j > i$  and  $Labeler_j$  is crashed, then  $report\_val(X^0(Labeler_j))[i] \in Bd(i)$ . (Recall that the report value of the ultimate event by the crashed  $Labeler_j$  is determined by the  $Update$  event in  $X^0(Labeler_j)$  and the value that it writes on  $Head(i)$ .)

**Lemma 10.17** *Suppose that  $L$  is a LABEL operation execution by  $Labeler_i$ . Assume that  $j > i$  and  $Labeler_j$  is crashed. Then  $diagonal(L)$  is different from  $d_0 = report\_val(X^0(Labeler_j))[i]$ . Hence  $d_0 \in Bd(i)$ .*

**Proof:** It is exactly for this lemma that the *LABEL* protocol contains in line **up**(b) a *Scan* of all *Head* registers of higher index. If  $L$  is a *Labeler<sub>i</sub>* operation execution,  $j > i$ , and *Labeler<sub>j</sub>* is crashed, then *Head*( $j$ ) is scanned in  $L$  and the diagonal number found in field *report*[ $i$ ] is added to variable *evade*. But this diagonal number is *report\_val*( $X^0(\text{Labeler}_j)$ )[ $i$ ], and so the lemma follows. ■

We shall define next, for every  $1 \leq i \leq N$ , a linear ordering  $<_i$  of  $Bd(i)$ , called the “stable ordering of  $Bd(i)$ ”. We will do it first when *Labeler<sub>i</sub>* is not crashed and then when it is crashed.

**Definition 10.18 (stable ordering of  $Bd(i)$ )**

Assume that *Labeler<sub>i</sub>* is not crashed, and let  $d \in Bd(i)$  be a bounded diagonal number in *Labeler<sub>i</sub>*. By definition,  $d$  is the diagonal number of only a finite number of operations in *Labeler<sub>i</sub>*. Consider the place of  $d$  in *trace*( $L$ ) where  $L$  is a *Labeler<sub>i</sub>* event that starts after  $m_1$ . Since  $d$  is never a new diagonal number of a *Labeler<sub>i</sub>* operation, the place of  $d$  can only decrease or remain in place when a new diagonal number is added by *PUT\_LAST* (execution of line 2(e)). Thus there is a moment  $\mu(i)$  (later than  $m_1$ ) after which the place of every diagonal number in  $Bd(i)$  is stable: namely, it will never decrease or increase. Consequently, there is a linear ordering  $<_i$  of  $Bd(i)$  such that if  $L$ , by *Labeler<sub>i</sub>*, begins after  $\mu(i)$  then  $Bd(i)$  is an initial segment of *trace*( $L$ ) and *trace*( $L$ ) restricted to  $Bd(i)$  equals  $<_i$ .

If *Labeler<sub>i</sub>* is crashed, then  $Bd(i) = \mathbf{diagonal\_number}$  and we let

$$<_i = \text{trace}(X^0(\text{Labeler}_i))$$

be the ordering of the diagonal numbers given by the ultimate *UpDate* on *Head*( $i$ ). Recall that if *Labeler<sub>i</sub>* is crashed, then *diagonal*( $X^0(\text{Labeler}_i)$ ) is the last diagonal number in  $<_i$  (this is guaranteed by the requirement on the **head\_type**).

**Definition 10.19** *We say that  $t$  in **diag\_order** is “stable for  $i$ ” iff  $Bd(i)$  is an initial segment of  $t$  and  $<_i$  is the restriction of  $t$  to  $Bd(i)$ . In details, the requirements are that if  $a$  and  $b$  are diagonal numbers such that  $a < t b$  and  $b \in Bd(i)$  then  $a \in Bd(i)$  as well, and if both  $a$  and  $b$  are in  $Bd(i)$  then  $a < t b$  iff  $a <_i b$ .*

Let  $m_2$  be a moment later than every  $\mu(i)$ . If  $L$  in *Labeler<sub>i</sub>* starts after  $m_2$  then *trace*( $L$ ) is stable for  $i$ , and *diagonal*( $L$ )  $\notin Bd(i)$ . Let  $m'_2 > m_2$  be

such that every non-crashed  $Labeler_i$  process has a  $LABEL$  event  $M(i)$  that begins after  $m_2$  and ends before  $m'_2$ . Any operation execution that begins after  $m'_2$  is said to be “stable”.

Let  $m_3 > m'_2$  be such that every non-crashed index  $i$  has a  $LABEL$  event in  $Labeler_i$  that begins after  $m'_2$  and ends before  $m_3$ . We shall prove that any moment  $m \geq m_3$  satisfies the self-stabilizing timestamp axiom  $P_{cts}(m)$  of 9.2.1.

Choose any moment  $m$  after  $m_3$ . The following property can be deduced.

If  $S$  is a  $SCAN$  event that begins after  $m$ , then  $\Omega_i(S)$  is either (8)  
the ultimate  $X^0(Labeler_i)$  (if  $Labeler_i$  is crashed) or else  $\Omega_i(S)$   
is a stable  $Labeler_i$  event (one that begins after  $m'_2$ ).

**Definition 10.20** Let  $1 \leq i \leq N$  be a  $Labeler$  index and  $s = \langle s[1], \dots, s[i] \rangle$  be a sequence of length  $i$  of diagonal numbers (that is, a member of  $\mathbf{report\_type}_i$ ).

**bounded:** We say that  $s$  is “bounded” if  $s[k] \in Bd(k)$  for every  $1 \leq k \leq i$ .

We proved in Lemma 10.17 that if  $Labeler_i$  is crashed then  $report\_val(X^0(Labeler_i))$  is bounded.

**active:** We say that  $s$  is “active” if

1.  $s[i] \notin Bd(i)$ , and
2. for every  $1 \leq k \leq i$ ,  $Labeler_k$  is crashed iff  $s[k] \in Bd(k)$ , and in case  $Labeler_k$  is crashed then  $s[k] = diagonal(X^0(Labeler_k))$ .

**Lemma 10.21** If  $L$  is a stable  $LABEL$  event, then  $s = report\_val(L)$  is active.

**Proof:** Assume that  $L$  by  $Labeler_i$  is stable (i. e.  $m'_2 < begin(L)$ ). Since  $m_1 < begin(L)$ ,  $diagonal(L) \notin Bd(i)$ . Say  $s = report\_val(L)$ . So  $s[i] \notin Bd(i)$ . For  $k < i$ ,  $s[k] = diagonal(E\_avoid(L, k))$  by Lemma 10.5. Say  $M = E\_avoid(L, k)$ . Assume  $Labeler_k$  is not crashed. By the definition of  $m_2$ , we know that  $M(i) \rightarrow M$ , and hence  $M$  begins after  $m_2$ . So  $diagonal(M) \notin Bd(k)$  as required. ■

**Lemma 10.22** Let  $s = \langle s[1], \dots, s[i] \rangle$  and  $t = \langle t[1], \dots, t[j] \rangle$  be in  $\mathbf{report\_type}_i$  and  $\mathbf{report\_type}_j$  respectively. Let  $u = \langle u_1, \dots, u_N \rangle$  be an arbitrary sequence of  $N$  stable  $\mathbf{diag\_order}$  (where  $u_i$  is stable for  $i$ ).

1. If  $s$  is bounded and  $t$  is active then  $s \text{ LEX}(u) t$ .
2. If both  $s$  and  $t$  are bounded then for any  $v = \langle v_1, \dots, v_N \rangle$  another sequence of  $N$  stable **diag\_order**

$$(s \text{ LEX}(u) t) \text{ iff } (s \text{ LEX}(v) t).$$

**Proof:** To prove the first item observe that  $t$  cannot be an initial segment of  $s$  because  $t$  is active and hence  $t[j] \notin \text{Bd}(j)$ . So either  $s$  is an initial segment of  $t$  or else there is a first place of disagreement and then  $s \text{ LEX}(u) t$  because the orderings of  $u$  are stable.

We explain item 2: since both  $s$  and  $t$  are bounded,  $s[k], t[k] \in \text{Bd}(k)$  for every  $k \leq \min\{i, j\}$ . So  $u_k$  and  $v_k$  agree on the ordering of  $s[k]$  and  $t[k]$  (and they both agree with  $\langle_k$ ). ■

We are ready now to extend  $\Rightarrow$  over the set of ultimate events of crashed processes as well. For this we define a relation  $\Rightarrow'$  as follows.

1. If  $L_1$  and  $L_2$  are both normal *Labeler* events, then  $L_1 \Rightarrow' L_2$  iff  $L_1 \Rightarrow L_2$ . So  $\Rightarrow'$  extends  $\Rightarrow$ .
2. If  $\text{Labeler}_i$  is crashed and  $L$  is a normal *LABEL* event, then

$$X^0(\text{Labeler}_i) \Rightarrow' L.$$

3. If both  $\text{Labeler}_i$  and  $\text{Labeler}_j$  are crashed, then let  $b = (\langle_1, \dots, \langle_N)$  be the sequence of orderings, where each  $\langle_n$  is the stable ordering defined above on  $\text{Bd}(n)$ . Define

$$X^0(\text{Labeler}_i) \Rightarrow' X^0(\text{Labeler}_j)$$

iff

$$\text{report\_val}(X^0(\text{Labeler}_i)) \text{ LEX}(b) \text{report\_val}(X^0(\text{Labeler}_j)). \quad (9)$$

If  $\text{Labeler}_i$  and  $\text{Labeler}_j$  are crashed and  $s = \text{report\_val}(X^0(\text{Labeler}_i))$ ,  $t = \text{report\_val}(X^0(\text{Labeler}_j))$ , then  $s$  and  $t$  are bounded **report\_type** vectors and the ordering defined here corresponds to that of Lemma 10.22(2).

It is obvious that  $\Rightarrow'$  is a linear ordering of its domain.

**Lemma 10.23** *Let  $S$  be a SCAN operation that begins after  $m$ . Then, for every indexes  $j \neq i$ , if  $Labeler_i$  is crashed then*

$$i \triangleleft(S) j \text{ iff } \Omega_i(S) \Rightarrow' \Omega_j(S). \quad (10)$$

**Proof.** Let  $S$  be a SCAN operation execution that begins after  $m$ . Let  $h = h^S$  be the value of variable  $h$  in  $S$  after the *Scan* action of line 1. (So  $h[k]$  is the value read in  $Head(k)$ .) Let  $basis = basis^S$  be the value of that variable after line 1 is executed in  $S$ . (So  $basis[k] = h[k].trace$ .) The SCAN protocol defines

$$i \triangleleft(S) j \text{ iff } h[i].report \text{ LEX}(basis) h[j].report. \quad (11)$$

We claim that  $basis$  is a sequence of stable orderings ( $basis[i]$  is stable for  $i$ ). If  $Labeler_i$  is crashed then  $\Omega_i(S) = X^0(Labeler_i)$  and hence  $basis[i] = trace(X^0(Labeler_i)) = \triangleleft_i$ . If  $Labeler_i$  is not crashed then  $\Omega_i(S)$  is a  $Labeler_i$  event that starts after  $m'_2$  and is thence stable. So  $h[i].trace$  is stable for  $i$ .

If both  $i$  and  $j$  are crashed, the proof of (10) follows from (11), Lemma 10.22, and (9). If  $i$  is crashed but  $j$  is active, then  $\Omega_i(S) = X^0(Labeler_i) \Rightarrow' \Omega_j(S)$  and we must prove that  $i \triangleleft(S) j$ , but this follows from Lemma 10.22(1) (since  $h[i].report$  is bounded, by Lemma 10.17, and  $h[j].report$  is active, by Lemma 10.21).

The following Theorem is the heart of our correctness argument.

**Theorem 10.24 (global coherence)** *Suppose that  $S$  is a SCAN event beginning after  $m$ ,  $L_1 = \Omega_i(S)$  and  $L_2 = \Omega_j(S)$ . Then  $L_1 \Rightarrow' L_2$  if and only if  $i \triangleleft(S) j$ .*

**Proof:** Since  $\Rightarrow'$  is a linear ordering, it suffices to prove that  $L_1 \Rightarrow' L_2$  implies  $i \triangleleft(S) j$ . Assume  $L_1 \Rightarrow' L_2$ . If  $Labeler_i$  is crashed, then  $i \triangleleft(S) j$ , by the previous lemma. So assume that  $Labeler_i$  is not crashed. Since  $L_1 \Rightarrow' L_2$ ,  $Labeler_j$  is not crashed. Thus both processes are not crashed,  $L_1$  and  $L_2$  are stable LABEL executions, and  $L_1 \Rightarrow' L_2$  translates back to  $L_1 \Rightarrow L_2$ .

We are going to use the following notation: for any index  $m$  and normal LABEL  $L$  in  $Labeler_m$

$$E\_reports(L) = \langle E\_report(L, 1), \dots, E\_report(L, m) \rangle,$$

and

$$diagonals(E\_reports(L)) = \langle diagonal(E\_report(L, 1)), \dots, diagonal(E\_report(L, m)) \rangle.$$

Then  $diagonals(E\_reports(L)) = report\_val(L)$  by Lemma 10.5.

Following the definition of  $\Rightarrow$ , there can be two reasons for  $L_1 \Rightarrow L_2$ . Assume first the “else” clause of Definition 10.14, that is that  $i < j$  and the sequence  $r_1 = E\_reports(L_1)$  is an initial segment of  $r_2 = E\_reports(L_2)$ . Then  $diagonals(r_1)$  is an initial sequence of  $diagonals(r_2)$ , and then  $i \triangleleft(S) j$  (since  $diagonals(r_n) = report\_val(L_n)$  for  $n = 1, 2$ ).

Next assume that the reason for  $L_1 \Rightarrow L_2$  is that for some  $k \leq \min\{i, j\}$  and  $V_1, V_2$  of index  $k$ ,  $V_1 = E\_report(L_1, k)$ ,  $V_2 = E\_report(L_2, k)$ , and  $V_1 \rightarrow V_2$ ; and for all  $\ell < k$ ,  $E\_report(L_1, \ell) = E\_report(L_2, \ell)$ . So, the two sequences:

$$diagonals(E\_reports(L_1))$$

and

$$diagonals(E\_reports(L_2)),$$

do not differ before the  $k$ -th entries (which are  $diagonal(V_1)$  and  $diagonal(V_2)$ ). Assume for a contradiction that  $j \triangleleft(S) i$ . So

$$diagonal(V_2) \preceq_k diagonal(V_1),$$

for  $\preceq_k = trace(V^*)$ ; where  $V^* = \Omega_k(S)$ . Lemma 10.11 may be applied to  $L_2 = \Omega_j(S)$ , and  $V_2 = E\_report(L_2, k)$  to yield  $V_2 \rightarrow= V^*$ . By Lemma 10.8 it follows that for some  $V_3$  of index  $k$ ,  $V_2 \rightarrow= V_3 \rightarrow= V^*$  and  $diagonal(V_1) = diagonal(V_3)$ . The Report Lemma 10.13 applies to  $L_1$ ,  $V_1 = E\_report(L_1, k)$ , and  $V_1 \rightarrow V_3$ , and it implies that  $L_1^+$  exists and  $L_1^+ \rightarrow W_2$  where  $W_2 = E\_avoid(V_3, i)$ .

Clearly,  $L_1^+ \rightarrow S$  is impossible, because  $L_1 \rightarrow L_1^+ \rightarrow S$  is in contradiction to  $L_1 = \Omega_i(S)$ . Hence  $begin(S) \leq end(L_1^+)$  and so  $begin(S) < begin(W_2)$ . If  $w^*$  is the *UpDate* operation in  $V^*$ , then  $begin(w^*) < begin(S)$  (because  $V^* = \Omega_k(S)$ ). As  $V_3 \rightarrow= V^*$ , we have  $r \rightarrow w^*$  where  $r$  is the read of  $LL(i, k)$  in  $V_3$ . So  $r \rightarrow S$  and hence  $r \rightarrow W_2$ . But  $\omega(r) \in W_2$  by definition of  $W_2 = E\_avoid(V_3, i)$  and this contradicts the regularity of  $\omega$ . ■

## 11 Conclusion

Our paper contains two new algorithmic results: a self-stabilizing weak snapshot, and an efficient self-stabilizing timestamp. We continue here the work of Dwork et al. [17] who defined the notion of weak snapshot and showed its usefulness. In fact, we define here a slightly stronger type of weak snapshot: we define a set of properties (the *wss* intermediate axioms of Section 3), and

we prove that they imply the weak snapshot properties of [17]. Our weak snapshot algorithm is not only self-stabilizing, but actually satisfies these stronger properties and the usefulness of this is shown in the second part of the paper in which a self-stabilizing timestamp protocol is investigated which uses (stronger) weak snapshots as building blocks.

As defined in [17] a weak snapshot for  $n$  processes  $P_1, \dots, P_n$  is an array of registers  $R_1, \dots, R_n$  where  $P_i$  writes some data values on  $R_i$  which is read by all processes. The write operation is called *Update*, and the *Scan* operation is a read of all registers that has stronger properties than a simple collect but is not as strong as an atomic snapshot. The weak snapshot algorithm of [17] employs single writer multiple reader atomic registers for interprocess communication. Each *Scan* and *Update* operation requires  $O(n)$  accesses to registers of size  $O(n + V)$  bits, where  $V$  is the number of bits required for the data values written and read by the processes. Our snapshot algorithm has the same computational complexity of  $O(n)$  accesses to registers per *Scan/Update* operation. Our registers, however, are smaller and simpler. We have single writer (atomic and regular) registers of size  $O(n)$ , and safe registers of size exactly  $V$  that carry just the data values. This separation of values from control is especially useful since safe registers are considered to be of the simplest possible kind.

The main issue in this paper is self-stabilization, and the weak snapshot algorithm that we define here is self-stabilizing. It is not inconceivable that the algorithm of [17] can be made to be self-stabilizing, but since both the description of that algorithm and its correctness proof depend on the Traceable Use abstraction of Dwork and Waarts ([15] and [16]), one would have first to find a self-stabilizing Traceable Use abstraction and prove that the resulting algorithm is self-stabilizing. In contrast, our presentation of the weak snapshot algorithm and its correctness proof are self contained.

As explained in the Introduction Section, the main advancement of this paper is in reducing the complexity of the self-stabilizing timestamp of [5] from  $O(N^2)$  read/write accesses per *SCAN/LABEL* operation to  $O(N)$  accesses. Yet not in every aspect our present paper improves on [5]: while the *LABEL* operation there takes just  $< 5N$  read accesses, in the present paper it needs  $O(N)$  accesses. Moreover, the scanners in [5] do not write at all, they are totally silent. Here, each scanner also writes (on a register of size  $O(N)$  bits).

Our self-stabilizing timestamp can also be compared to the timestamping algorithm of [17]. (As mentioned, the authors of [17] do not consider self-

stabilization and it is not unreasonable to think that it is probably not self-stabilizing). Both algorithms take  $O(N)$  read/write accesses per operation. The registers of [?] are of size  $O(N+V)$  bits while our registers here are of size  $O(N \log N + V)$  bits. We note, however, that while [?] uses atomic single writer registers, we use here both atomic and safe registers: our atomic registers are of size  $O(N)$  and our safe registers are of size  $O(N \log N + V)$  bits.

## References

- [1] U. Abraham. On interprocess communication and the implementation of multiwriter atomic registers. *Theoretical Computer Science* 149, 2, 257-298, 1995.
- [2] U. Abraham. On system executions and states. *Journal of Applied Intelligence*, 3, 17-30, 1993.
- [3] U. Abraham. What is a State of a System? (an outline). in M. Droste and Y. Gurevich ed. *Semantics of Programming Languages and Model Theory*, Vol. 5 of Algebra, Logic and Applications Series, Gordon and Breach, 1993.
- [4] U. Abraham. *Models for Concurrency*, Gordon and Breach, 1999.
- [5] U. Abraham. Self-stabilizing timestamps. *Theoretical Computer Science* 308(1-3), 449-515, (2003).
- [6] U. Abraham, S. Ben-David and M. Magidor. On Global-Time and inter-process communication, in M. Z. Kwiatkowska et al., eds., Proceedings of the International BCS-FACS Workshop on Semantics for Concurrency, (Leicester, 1990) 311-323 Workshops in Computing, a Springer Series edited by C. J. van Rijsbergen.
- [7] U. Abraham, S. Dolev, T. Herman, and I. Koll. Self-Stabilizing  $\ell$ -Exclusion. *Theoretical Computer Science* Vol. 266, pp. 653-692, (2001).
- [8] U. Abraham and T. Pinhas. Exercises in Style (Alpha Specifications). *Fundam. Inform.* 54(2-3) 107-135, (2003).
- [9] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory, *J. ACM*, 40 (1993), pp. 872-890.

- [10] J. Anderson. Composite registers, *Distrib. Comput.*, 6 (1987), pp. 141–154.
- [11] H. Attiya, M. Herlihy, and O. Rachman. Efficient atomic snapshots using lattice agreement, in *Proc. 6th International Workshop on Distributed Algorithms*, Lecture Notes in Comput. Sci. 647, Springer-Verlag, New York, 1992, pp. 35–53.
- [12] H. Attiya and O. Rachman, Atomic snapshots in  $O(n \log n)$  operations, in *Proc. 12th ACM Symposium on Principles of Distributed Computing*, Ithaca, NY, ACM, New York, 1993, pp. 29–39. And journal version: *SIAM J. Comput.* 27(2): 319-340 (1998).
- [13] B. Bloom. Constructing two-writer atomic registers. 6th ACM Symp. on Principles of Distributed Computing, pp. 249 - 259. 1987.
- [14] D. Dolev and N. Shavit. Bounded concurrent time-stamp systems are constructible. *Proceedings of the 21st ACM Symposium on Theory of Computing*, pages 454–465. 1989. Final version: Bounded Concurrent Time-Stamping, *Siam J. Comput.* Vol. 26, 2, pp. 418–455, 1997.
- [15] C. Dwork and O. Waarts. Simple and efficient bounded concurrent time-stamping (or bounded concurrent timestamping are comprehensible!). *Proceeding of the 24th ACM Symposium on Theory of Computing*, pages 655–666. 1992.
- [16] C. Dwork and O. Waarts. Simple and Efficient Bounded Concurrent Timestamping and the Traceable Use Abstraction. *Journal of the ACM*, Vol. 46 No. 5 pp. 633-666, 1999.
- [17] Cynthia Dwork, Maurice Herlihy, Serge Plotkin, and Orli Waarts, Time-Lapse Snapshots. *Siam J. Comput.* Vol. 28, 5 pp. 1848–1874 (1999).
- [18] A. Israeli and M. Li. Bounded Time Stamps. *Proceedings of the 28th annual IEEE Symposium on Foundations of Computer Science*, pp. 371–382. 1987.
- [19] A. Israeli and M. Li. Bounded time-stamps. *Distributed Computing*, Vol. 6, No.4, pp. 205-209, 1993.

- [20] A. Israeli and M. Pinhasov. A concurrent Time-Stamp scheme which is linear in time and space. In A. Segall and S. Zaks, editors, *Distributed Algorithms: Sixth International Workshop (WDAG '92, Haifa, Israel, November 1992)*, Vol. 647 of *Lecture Notes in Computer Science*, pp. 95-109, Springer-Verlag, 1992.
- [21] Lefteris M. Kirousis, Paul Spirakis, and Philippas Tsigas. Simple atomic snapshots: A linear complexity solution with unbounded time-stamps. *Information Processing Letters*, 58(1):47-53, 8 April 1996.
- [22] L. Lamport. On Interprocess Communication, Part I: Basic formalism, Part II: Algorithms. *Distributed Computing*, Vol. 1, pp. 77 - 101. 1986.
- [23] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers* Addison-Wesley, 2002.
- [24] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer 1992.