On the Wait-Free Power of Iterated-Immediate-Snapshots *

(Extended Abstract)

Eli Gafni

(eli@cs.ucla.edu)

UCLA Computer Science Department

Los Angeles, CA 90024

January 9, 1998

Abstract

The Iterated Immediate Snapshot model, introduced recently, has proved its usefulness. Its rich structure provided alternative simple derivation of past results, and its topological equivalence to the Euclidean Unit Simplex provides insights into new problems. Central to the model usefulness is the fact that it was shown to be non-blocking equivalent to the Atomic-Snapshot memory model and as a result equivalent to the message-passing system. Yet, to investigate long-lived tasks we need a stronger notion of equivalence, wait-free equivalence, rather than just non-blocking one. This paper provides a new generalized definition of wait-free implementation and correspondingly a novel algorithm that establishes the wait-free equivalence of the models. As a by product, it motivates an alternative elegant characterization of tasks that are t-resiliently solvable in the AS model.

*Contact Author: Eli Gafni. Submitted for Long Presentation Track, PODC98

1 Introduction

Using transformations between spaces to solve mathematical problems is as old as modern mathematics. Convolution in the time domain is solved by transforming it into multiplication in the frequency domain through the back and forth Fourier transforms. Problems with real time-varying trigonometric functions are transformed into problems in the complex plane, etc. The idea of dealing with a problem by transforming it to a problem in a model different that the one it was posed in, is omni-present in distributed computing. Message passing-problems can be transformed into shared-memory problems [1]. Shared-memory problems can be transformed into Atomic-Snapshot (AS) shared-memory problems, etc.

The Iterated Immediate Snapshots (IIS) model was recently introduced [2], to derive a simple constructive characterization of problems that are wait-free solvable in read-write shared memory. In the short time since its introduction it was used in [3, 4]. Essentially it has two advantages over other equivalent models. First, algorithms stated in the model are forced into the template of communication-closed-layers [5] and are thus easy to prove correct. Second, the model has an explicit embedding in Euclidean space and thus allows all the topological machinery of this most natural space to be brought to bear on a problem. In a forth coming paper [4] we show that it gives rise to the notion of "Geometric-Protocols," that considerably simplifies the task of protocol design. Thus it would be beneficial if problems in the AS model could be transformed into problems in the IIS model.

The generic problem we face, and which is in our knowledge addressed for the first time, is that the notions of what constitutes a "step," or the notion of a "faulty" processor in the IIS model, are not that clear. In all the transformations encountered in the past there was a natural interpretation to what constitutes a step or a fault. The IIS model is in essence a mathematical object far removed from any "real" object to justify one definition or the other. Our contention is that it therefore should be treated as such and various definitions associated with it justified solely by their utility. The utility of the IIS model is as a vehicle to solve problems in the AS model.

To solve a problem Π in the Atomic-Snapshot (AS) memory model [6] using the IIS model, we need two transformations. A transformation $T_{IIS \leftarrow AS}$ that takes runs in the AS model and returns runs in the IIS model, and a transformation $T_{AS \leftarrow IIS}$ that takes runs in the IIS model and return runs in the AS model.

The utility of these transformation depends on the properties of their composition when applied to any run $r \in AS$, namely, $r' = T_{AS \leftarrow IIS}(T_{IIS \leftarrow AS}(r))$. The transformations are useless if the set of participating processors in r and r' are not identical. The subtler question is with regard to faultiness.

A pair of transformations that does not preserve faultiness (either by reviving a faulty processor or failing a correct one) is useful, but to a limited extend. If we want to use the IIS to solve a problem posed in the 1-resilient AS model, or to solve non-terminating problems (e.g. Mutual-Exclusion) wait-free, we should better have transformations that preserve faultiness. The transformation provided in [2] did not preserve faultiness but preserved the participating sets and as such were good for establishing the utility of IIS to solve 1-shot tasks in the AS. They did not establish the utility of IIS to solve problems in any restriction of the AS model, like the t-resilient AS model, or the utility of the IIS model to solve a non-terminating problems wait-free.

This paper provides a pair of transformations that preserves faultiness. These transformations *induce* the notion of faultiness in the IIS model. A processor P_i in a run $r_{IIS} \in IIS$ is faulty if and only if P_i is faulty in $T_{AS \leftarrow IIS}(r_{IIS})$. Since the pair of transformations preserves faultiness it follows that P_i is faulty in $r_{AS} \in AS$ if and only if it is faulty in the induced sense in $T_{IIS \leftarrow AS}(r_{AS})$.

Thus, to solve a task 1-resiliently in the AS model, we solve the task with respect to the induced set of runs in the IIS model. I.e. if the set of 1-resilient runs in the AS model is S, we solve the task in the IIS model with respect to the set $T^{-1}_{AS \leftarrow IIS}(S)$. To solve the original problem we take any run $r \in AS$ and assign to it the output assigned to the run $T_{AS \leftarrow IIS}(r) \in IIS$. By the observation in the preceding paragraph we have that $T_{AS \leftarrow IIS}(r) \in S$, and therefore has an output assigned to it.

The paper is organized as follows. Section two contains a short introduction to the two computational models of this paper, and few results with respect to the induced notion of faultiness in IIS. Section three present a simple amendment to the transformation in [2] to make it a wait-free transformation from IIS to AS. The novel part of the paper is the Red-Yellow-Green algorithm introduced in section four. After introducing it and discussing it properties, we further illuminate it, by using it to provide a new alternative characterization of tasks which are solved t-resiliently, and we then use it to provide a wait-free transformation from AS to IIS. We conclude in section five. A precise close to implementation spec of the Red-Yellow-Green algorithm appears in an Appendix.

2 Model

In the AS and the IIS models we have $n+1$ processors $P_0, P_1, ..., P_n$. In the AS model we have a single shared memory consisting of $n+1$ cells $C_0, C_1, ..., C_n$ initialized to $\perp$. Processor P_i writes exclusively into C_i and reads atomically all cells. In the IIS model we have an infinite sequence of memories $M_1, M_2, ...$ Computation proceeds by processor P_i writing to cell C_i in M_1 then reading M_1 in an Immediate-Snapshot. It then writes the cell C_i in M_2 and reads M_2 in an Immediate-Snapshot, etc, going from memory to memory in order, writing and reading once in each.

The Immediate-Snapshot is a restriction of the Atomic-Snapshot. In addition to returning a snapshot it requires that if P_i observes that P_j wrote, i.e. it read C_j to be different than $\perp$, then the set of cells P_j observed to have been written is a subset of the cells observed by P_i to have been written. Atomic-Snapshot just require a set containment relation between the two returns but does not restrict who should be subset of whom.

A run in the AS model is an infinite sequence of processor names, where the first appearance of a processor is interpreted as write, the second as read, the third as write, etc. A processor is faulty in a run if it appears in it only finitely many times, otherwise it is correct or non-faulty. It is sleeping in the run if it does not appear at all. If it is not sleeping it is then participating.

A run in the IIS model is an infinite sequence of ordered partitions of the set P of processors. For instance in case $n = 2$ the following is a run: $(0, 12)(1, 2, 0)(120), ...$ It describes the situation in which in M_1 processor P_0 observed only its cell have been written, while P_1 and P_2

observed all cells to have been written. In M_2 processor P_1 observed only its cell to have been written, processor P_2 observed cells C_1 and C_2, and P_0 observed all, etc.

We now define the induced notion of a faulty processor in a run in the IIS model and in later sections show that the transformations we introduced preserve faultiness going between the models.

Given a run r in the IIS we associate with it a directed graph G_r. There is a node (i, j), $i = 0, ..., n$, $j = 1, 2, ...$ per each processor in each partition. From (i, j) there is an outgoing edge to the node of type $(i, j-1)$, and outgoing edges to all nodes of type (k, j) provided P_k in partition j is either in the same part as P_i, or in a part that precedes it. If writing of a cell is a node, the interpretation of these edges is that P_i after writing C_i in M_j and reading, "knows" what was written in all the nodes reachable from (i, j).

We say that P_f is *faulty* in r if there exists j and i such that in G_r node (f, j) is not reachable by (i, k) for all k. I.e. a processor is faulty if it writes a cell and there exists another processor that never "knows" it. A processor which is not faulty is non-faulty or correct. A processor is *sleeping* in r if it is faulty and the j used equals 1. If a processor is not sleeping in r then it is *participating.*

We now establish few general properties relating to faultiness in IIS that will be helpful later.

Proposition 2.1 *If P_i is non-faulty and P_f is faulty in $r \in IIS$, then there exist j such that in G_r node (f, j) is not reachable by (i, k) for all k.*

Proof Suppose not. Then for all j there exists $k(j)$ such that (f, j) is reachable by $(i, k(j))$. Since P_i is non-faulty, each of its nodes is eventually reachable by nodes of all other processors, thus each of the nodes of P_f is, contradiction. $\Box$

Proposition 2.2 *For all $r \in IIS$ there exist i such that P_i is non-faulty.*

Proof Notice that by definition of G_r for all i, k, j we have either $((i, j), (k, j)) \in E_{G_r}$, or $((k, j), (i, j)) \in E_{G_r}$, or both. Thus for P_k to be faulty there must exist P_l such that eventually forever P_l appear in the partitions in-front of P_k. Since "appears in front eventually forever" is a directed acyclic relation, the proposition is proved. $\Box$

Proposition 2.3 *Let $r \in IIS$ be such that for G_r there exist k such that for all (i, j) if $S_{(i,j)}$ denotes the set of nodes reachable from (i, j) in k hops or less, then $|\{m | \exists w(m, w) \in S_{(i,j)}\}| \geq (n+1) - t$, then at most t processors are faulty in r.*

Proof Let $N_{(i,j)} = \{m | \exists w(m, w) \in S_{(i,j)}\}$. Define $N_{(i,io)} = \bigcap_{j=1,2,...} \bigcup_{k=j,j+1,...} N_{i,k}$. The set $N_{(i,io)}$ is the set of processors that P_i has a a direct edge to, infinitely often in the k-closure of G_r. Obviously, by the Pigeon-Hole principle, if one puts $n+1-t$ balls into $n+1$ distinct slots and repeats this ad-infinity, then at least $n+1-t$ slots will hold infinitely many balls, and thus $|N_{(i,io)}| \geq n+1-t$. Consequently, if P_i is non-faulty, then by Propositions 2.1, the set of faulty processors is of cardinality smaller or equal to t. By Proposition 2.2 there must exist a non-faulty processor, which establishes the proposition. $\Box$

3 Wait-free implementation of AS by IIS

The wait-free implementation of AS by IIS is a rather simple modification of the non-blocking implementation in [2]. The idea of the modification is as in [6], to let processor adopt an embedded snapshot. We first succinctly repeat the implementation in [7].

The idea there is to show the implementation in case processors have just a single WriteRead operation in which a processor writes increasing sequence numbers, and reads all sequence numbers of other processors. We have implemented an Atomic-Snapshot memory if for any two returned vectors of sequence numbers read, one is the same or majorizes the other.

The idea of the implementation is for processors to write their latest "estimate" vector to memory. A processor returns a snapshot for the AS, if in the IIS, in the last Immediate-Snapshot memory it read all vector were the same. If they are, it returns the vector, and in writing to the next memory, increases it own sequence number by 1. Otherwise, if the vectors it read are not the same, it takes the entry-wise maximum to be its estimate, which it writes into the next memory.

This implementation is not wait-free as the following run (1,2,0)(0,2,1)(1,2,0)(0,2,1)..., shows. In the odd positions in the sequence, processor 1 writes a new sequence number which processor 2 has never seen before (and therefore cannot return), and similarly, in the even cycles processor 0 introduces a new sequence number. Thus processor 2, although not faulty, will never return a value for the AS.

The simple idea, in line with [6], is for processors to write, together with their "estimate" vector, the last AS snapshot they have returned. A processor that sees an embedded snapshot that contains its own latest sequence number, returns this snapshot, and in the next memory, writes the entry-wise maximum of the previous memory and increases its own entry by 1.

To see that this implementation is wait free, notice that if the processor is not faulty, its new sequence number is bound to be picked up eventually by all non-faulty processors, and thus eventually, an embedded snapshot vector with the new sequence number is bound to be left behind, allowing the processor to always eventually return.

4 Wait-free implementation of IIS by AS

4.1 The Basic Technique

To achieve wait-freedom we usually employ the technique of "helping" by which processors perform work on behalf of other processors. We first remind the reader of the algorithm for one-shot immediate snapshot [7], into which we now want to introduce "help."

P_i ::
    for $j = n+1$ step -1 to 1 do
        place $token_i$ for $level_j$
        $read$:=read all processors k that placed $token_k$ for $level_j$
        if $|read| = j$
            then return $read$
                halt

od

The task a one-shot IS solves is for each processor starting with its ID as an input to return set of IDs and the sets comply with the IS requirement.

Suppose now we are in a t-resilient system. Then processors may wait until they observe at least $(n+1)-t$ inputs. We now ask to what degree they can wait-free coordinate themselves to obtain IS on behalf of the processors for which they know the input. We exemplify the idea of our algorithm by showing that we can wait-free obtain outputs on behalf of at least $(n+1)-t$ processor, the minimum size of the intersection of their inputs, without blocking a processor from obtaining an output for itself too. Further more, if P_i and P_j are in the intersection and $i<j$ than the snapshot returned on behalf of P_i will be strictly smaller than the one returned for P_j.

One would have thought that this situation is an ideal setting in which to apply the BG simulation [8, 9] in order to land processors an helping hand by others. The problem we faced is that blocking hindered progress. Indeed, a-posteriori, after we know that IIS is a wait-free alternative to read-write, then the algorithm we present is an alternative to to the BG simulation. In fact, it possesses all the properties of the BG simulation, with the additional property that if we associate particular code with a particular processor, then that code may only be blocked by the untimely death of that particular processor. Thus the Red-Yellow-Green algorithm we introduce may be useful in applications different than the one at hand.

Suppose each processor starts with a set I_i of processors IDs that includes its own ID. It wishes now to collect as many one-shot IS's as we can afford. The problem of course is that its work "on-behalf" of some other processor has to be coordinated with other processors trying to achieve something similar. What is reasonable to ask for?

Let S_i be the set of processor IDs for which processor P_i obtains a one-shot IS. Let P be the set of participating processors, below we show that we can satisfy the following requirements:

1. $P_i \in S_i$, (we do no worse than when $I_i = \{P_i\}$),
2. $|\bigcap_{P_i \in P} S_i| \geq |\bigcap_{P_i \in P} I_i|$, (we do no worse than $\forall P_i, P_j \in P,\ I_i = I_j$),
3. If $P_j \notin \bigcup_{P_i \in P} I_i$ than all IS snapshots returned contain $\perp$ for P_j, (this will prevent "default" solutions. The snapshot returned are compatible with P_j sleeping),
4. If $\forall i\ \exists r, q\ \ni,\ P_r, P_q \in I_i$ *or* $P_r, P_q \notin I_i$, and $r<q$ then $r \in S_q$ where S_q is the IS snapshot for P_q if it is returned.

Since we have seen that w.l.o.g. a one-shot snapshot is a set of indices we can now formulate the whole problem as a task without mentioning IS:

We are given $N = n+1$ processors $P_i,\ i = 0, 1, ..., n$. Each processor P_i has a set of indices $I_i \subseteq \{0, ..., n\},\ i \in I_i$ as input. Let P be the set of participating processors. Processor P_i outputs a set of sets of indices $O_i = \{S_{i,i_1}, S_{i,i_2}, ...\}$ (each $S_{i,m}$ is the set P_i returns on behalf of P_m) such that:

1. $S_{p,j} = S_{q,j},\ \forall p, q, j$, (sets obtained by different processors on behalf of the same processor agree),

2. $S_{i,i} \in O_i, \forall i$, (this is requirement 1 above)

3. $S_{p,i} \subseteq S_{q,j}$ or $S_{q,j} \subseteq S_{p,i}$, $\forall q, j, p, i$, (the AS requirement),

4. $p \in S_{i,j} \Rightarrow S_{p,p} \subseteq S_{i,j}$, $\forall p, i, j$, (the IS requirement)

5. $|\cap_{P_i \in P} O_i| \geq |\cap_{P_i \in P} I_i|$, (requirement 2 above),

6. $\forall i, j,\ S_{i,j} \subseteq \bigcup_{P_i \in P} I_i$, (requirement 3 above).

7. if $\forall i\ \exists r, q\ \ni,\ P_r, P_q \in I_i$ *or* $P_r, P_q \notin I_i$, and $r < q$ then if $S_{i,r} \in O_i$ and $S_{j,q} \in O_j$, then $S_{i,r} \subset S_{j,q}$ (requirement 4 above).

We solve this problem wait-free in the AS model. We call the algorithm the Red-Yellow-Green algorithm. We use the algorithm that implements one-shot IS in AS [7] above, only that we let processors "help" to move the virtual tokens of other processors. Tokens are moved until they get "stuck." Only after a token is stuck it returns back to be solely under the control of the owner processor.

Processors first announce their input to all by writing their input. At each level for each processor announced in some input and by each participating processor, we place now tokens of three colors. The first token is of red color to signify that the token was placed but the processor that placed it has no idea yet whether the level is full, meaning, that the number of processors associated with the tokens at the level is the same as the number of the level.

After placing a red token at a level on behalf of some processor P_k, it counts the number of processors on behalf of whom tokens have been placed at the level. If the number is below the level number, it considers the level not to be full and signifies it by placing a yellow token of behalf of P_k. It again observes the level and if it still not full it places a green token on behalf of P_k. A green token signifies that a red token for P_k may be put at one level down. Thus if a processor at any time observes a green token on behalf of P_k at the level, it can start placing tokens for P_k at the level below.

When a processor "works" on behalf of P_k and cannot continue in placing any more token on its behalf, then P_i considers P_k "stuck" and moves to help another processor. The next processor it picks is the one of the announced processor which it has not served, of lowest index. The reader is encouraged to observe already at this point that if all inputs are identical then tokens will be stuck at consecutive levels in relative order to their index. The second observation to make in this case is that all tokens will be observed "stuck on red." Namely, if tokens on behalf of P_k are stuck then there exist a level at which only red type token of behalf of P_k appear. In a global state in which all processors have observed P_k stuck on red we say that P_k is stuck on red.

In general, a processor P_k which is observed stuck on red may change and become "stuck on yellow," due to a processor that placed a red token for P_k, but was fast to observe the level before it became full. Similarly, "stuck on yellow" may change into "green." The main observation to make is that a processor that observes P_k to become stuck on red, has a guarantee that until such time that P_k takes control on its own destiny, no green token will be placed, since even a fast processor after placing the yellow token will observe the level to be full and quit handling P_k.

After P_i services all the processor announced, it then posts the set of processors T_i which it observed to be stuck on red. It reads in a snapshot all the T_j posted and accumulates it into a set of sets T. The processors which appear in all the T_j it observed, i.e. in $\bigcap T =: S_i$, is the processors for which it will return an IS. It returns an IS for P_k by reading all the processors with tokens at their behalf at the lowest level at which P_k has a token on its behalf. The only remaining problem to attend to is what if $P_i \notin \bigcap T$, since a processor has to return an IS for itself.

We distinguish two cases: If $P_i \in \bigcup T$ then it means that somebody observed P_i to be stuck on red and consequently, it will not move. Thus P_i just adds P_i to S_i, i.e. $S_i := S_i \bigcup \{P_i\}$. Otherwise it observes the token system again. If it sees itself stuck on red, it set $S_i := S_i \bigcup \{P_i\}$. Otherwise, it places a red token at a level below where it observed its lowest red token, and now plays the original game of our first algorithm until its red token is observed by it to reside at a level which is full. To prevent other processors from jumping in and helping it, before it starts it sets $permission_i := denied$. This signifies to other processors that simulating P_i is not permitted any more. When done, it set $S_i := S_i \bigcup \{P_i\}$ and return snapshot to processor$P_j \in S_i$ by observing the lowest level for with P_j has a red token and returning all the processors that have tokens at that level.

The crucial observation is the following Proposition.

Proposition 4.1 *The number of processors observed by P_i to be stuck on red is greater equal $|\bigcap_{P_j \in P} I_j|$.*

Proof For all pairs of processor P_q and P_r for which we have the property that for all i either $P_q, P_r \in I_i$ or $P_q, P_r \notin I_i$, then the moment that one of them will be announced, the other will be announced too. Thus if $r < q$, a level, say $level_m$, at which P_r is observed stuck, is filled up before any processor resorts to working on behalf of P_q. Since $level_m$ is observed already to be full and only now somebody may start simulating P_q, we conclude the P_q will never reach $level_m$, and will get stuck at a higher level.

Since at the least all the pairs of processors in $\bigcap_{P_j \in P} I_j$ possess the property of either appearing together or, vacuously, not at all, we obtain that no two processors in the intersection will get stuck at the same level.

At each level in which any processor is observed stuck there must be a processor which is stuck at red. This follows from the fact that the first red token which is the last token to make the level full, will be viewed as such and no yellow token will be placed for it.

Since we argued that the number of full levels is at least as big as the intersection, the result follows.

□

Corollary 4.1 *If P_q and P_r are such that for all i either $P_q, P_r \in I_i$ or $P_q, P_r \notin I_i$, and $r < q$ then P_q gets stuck at a level higher than P_r.*

4.2 characterization of t-resilient by the Red-Yellow-Green algorithm

In a t-resilient AS system when processors first write an then wait until they observe at least $(n+1)-t$ inputs, we have seen that we can wait-free solve the problem in the previous section with I_i being the set of processors observed. Imagine that what we actually did is simulate the behavior of processors in M_1 in an IIS system. We now continue to simulate them in M_2. We notice that we have again the same situation as in the beginning, i.e. each processor has a set of indices with the intersection at least of size $(n+1)-t$, and moreover, for each processor in S_i processor P_i knows its output from M_1 in order to start simulating it in M_2.

We want the run that the simulation produces to be a t-resilient run. For that we want processor in the simulated IIS to observe as many other processors as we can, as to reduce the potential of creating a faulty processor eventually. For that reason in the simulation of M_2 the order that we consider processors P_r and P_q will be in reverse order to the size of a snapshot they returned from M_1. If P_r returned a small snapshot, and P_q returned a bigger one, we will drop the tokens for P_q before P_r and thus P_r will see all that P_q has seen in M_1.

The crucial observation here is that if we follow this reverse order any simulated processor past M_2 has observe directly or indirectly at least $(n+1)-t$ processors. Since we now can repeat it indefinitely, by Proposition 2.3 with $k=3$ it follows that we produced a t-resilient run.

The reason for why the observation holds is that the size of each S_i is larger $(n+1)-t$. Thus the first token any processor start dropping in M_2 is on behalf of a processor that has seen at least $(n+1)-t$ processors in M_1. Thus, every processor in M_2 will see at least one of these first processors, and the observation follows.

We thus also see that we have t-resiliently implemented the second Standard Chromatic Subdivision [10], and since we proceed wait-free we thus obtain the following elegant characterization of resiliency that can also be derived by the characterization in [11]. There is was observed that for any task that is solved t-resiliently there exist an algorithm where for each processor there is only a single wait statement following its first write, where afterwards it proceeds wait-free. The framework here leads us to observe an alternative statement.

Theorem 4.2 *A task $(\mathcal{I}, \mathcal{O}, \Delta)$ is solvable t-resiliently iff $(\mathcal{I}', \mathcal{O}, \Delta')$ is solvable wait-free. Where $\mathcal{I}' = SCS^2(\mathcal{I})-$faces of dimension less than $(n+1)-t$, and for a simplex $s \in I'$, $\Delta'(s)$ is the projection of $\Delta(carrier(s))$ on $colors(s)$.*

4.3 Wait-Free Implementation of IIS by AS

In the previous subsection we have seen that we can transform the set of t-resilient runs in the AS model, to t-resilient run in the IIS model. But the transformation depended on the a-priori knowledge of t. Moreover it did not preserve individual faultiness. What we are seeking now is a single transformation that does not refer to t, which will transform runs in such a way that if it happens that the run is t-resilient the corresponding run will be t-resilient too, and it will preserve faultiness (non-faultiness). For that we use the Red-Yellow-Green algorithm.

Imagine that what the processors in the simulated IIS do, is to simulate AS by the sequence number vector facility of the third section. As before we let all processors simulate all other. Thou, there are things only the owner processor can do. First, after a WriteRead operation of P_i terminates at M_j, no processor may simulate P_i past M_j. It is up to P_i itself to start placing tokens at M_{j+1} by that indicating it invokes a new WriteRead operation. I.e. P_i is then blocked to all but itself. If P_i is non-faulty our rules will eventually lead it to start placing tokens for itself at M_{j+1}, allowing other processors now to lend a helping hand in terminating the operation thus started.

Secondly, P_i may be blocked from being simulated because its last token is in a level which is full, and it is observed stuck on yellow. Thus a processor according to a rule to be specified, picks another processor and tries to simulate it until it blocked from simulating it either on account of getting stuck on yellow or on account of terminating the current WriteRead operation on its behalf. As a result of the latter case the priority of that simulated processor may change, and the simulator picks up the next highest priority processor to simulate which is not blocked to it.

The rule of which processor to try to simulate next is extremely simple: If P_i observes P_q to complete two WriteRead operations while at the same time P_r did not complete a single one, then simulating P_r takes priority over simulating P_q. Ties are broken by an arbitrary common rule. Thus a processor maintains an ordered list of the other processors. It goes through the list in decreasing order of priority until it finds the first processor that is not blocked for it. It then simulates it until blocked, or until terminating its current WriteRead operation. It then scan the date structure, updates its priority list and starts again.

Thus the identity of a simulating processor matters only through the fact that as a simulated processor it may be in a situation in which its simulation is blocked to all but itself. In fact, the simulated self is never blocked for the simulator, in contrast with the BG simulation.

To see that if P_i is not faulty in the simulating process then it is not faulty in the resultant simulated run we make the following observations. Suppose by way of contradiction that F is the set of processors which is non-faulty in the simulating process (AS) but faulty in the simulated run (IIS). Obviously, eventually all simulated processors in F will have a priority higher than processors which are non-faulty. Thus if $P_i \in F$ and P_i is not blocked to itself, eventually the simulated P_i will indicate that it has started a new WriteRead operation. This will eventually be true for all processors in F.

Thus the only reason processors in F do not make progress in the simulated run is that they get blocked stuck on yellow. But if we are at a point in time after which no correct processor sees any processor in F then F is like a system on it own. In a system there is at-least one processor that is not stuck on yellow, namely, the last processor to get stuck. Thus eventually all simulators are engaged in continuously simulating processors in F, which will lead at least one processor in F to terminate its WriteRead. Contradiction.

5 Conclusions

We have established an important relation between IIS and AS. While it was known that they have the same power to solve tasks, here we have shown that they are identical models for any application. Since we envision the IIS as an extremely useful mathematical tool, such a result is of significance. Moreover we have hinted at how our simulation is a better alternative then the BG simulation. In fact, it is a structured alternative to the convergence algorithm in [2], an algorithm which in our opinion is the essence of what a protocol is.

Acknowledgment: I am in debt to Hagit Attiya for alerting me to the possibility of a wait-free equivalence between IIS and AS.

References

[1] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *Journal of the ACM*, vol. 42, pp. 124–142, Jan. 1995.

[2] E. Borowsky and E. Gafni, "A simple algorithmically reason characterization of wait-free computations," in *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, pp. 189–198, 1997.

[3] G. Hoest and N. Shavit, "Towards a topological characterization of asynchronous complexity," in *To appear in the Proceedings of the 16th ACM Symposium on Princip les of Distributed Computing*, 1997.

[4] E. Gafni, "Geometric Protocol," tech. rep., University of California, Los Angeles, Jan 1998.

[5] T. E. Elrad and N. Francez, "Decomposition of distributed programs into communication-closed layers," *Science of Computer Programming*, vol. 2(3), 1982.

[6] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merrit, and N. Shavit, "Atomic Snapshots of Shared Memory," in *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pp. 1–13, 1990.

[7] E. Borowsky and E. Gafni, "Immediate Atomic Snapshots and Fast Renaming," in *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, pp. 41–51, 1993.

[8] E. Borowsky and E. Gafni, "Generalized FLP Impossibility Result for t-Resilient Asynchronous Computations," in *Proceedings of the 25th ACM Symposium on the Theory of Computing*, pp. 91–100, 1993.

[9] N. Lynch and S. Rajsbaum, "On the borowsky-gafni simulation algorithm," in *ISTCS96*, 1996.

[10] M. Herlihy and N. Shavit, "A Simple Constructive Computability Theorem for Wait-free Computation," in *Proceedings of the 26th ACM Symposium on the Theory of Computing*, 1994.

[11] E. Borowsky, "Capturing the Power of Resiliency and Set Consensus in Distributed Systems," tech. rep., University of California, Los Angeles, Oct 15, 1995.

6 Appendix

Each processor P_i creates a shared bank of tokens of type $T(j, color, owner)$, $\forall j = 0, ...n$, $color \in \{red,\ yellow,\ green\}$, $owner \in \{public, private\}$. The first subscript j signifies that these token are in order to "simulate" P_j. Processors place shared tokens in a level system that contains $n+1$ levels $level.n+1, ..., level.1$. There is an additional starting level called $level.start$ in which tokens that appear as input to participating processors are announced. Finally, to capture the asynchronous nature of token placement we have a $tentative.level.m$ corresponding to each $level.m$. Tokens are placed at the tentative level, as a result of "reading," and are later moved to the corresponding level as a result of "writing." "Signature"" and "Status" section of the code are omitted.

Transitions:

place-$T(j, green)$*-at-level.start*	processors announce their inputs
Precondition:	
$j \in I_k$	
$T(j, green) \notin level.start$	by placing green tokens at
Effect:	
$level.start := level.start \bigcup \{T(j, green)\}$	level.start
tentative-place-$T(j, red)$*-at-level.m*	processors decide whether
Precondition:	a token can be pushed one level down
$m \neq n+1$ or $T(j, red) \in level.start$	case $m = n+1$
$permission_j = granted$	push by any processor
$m = n+1$ or $T(j, green) \in level.m+1$	case $m \neq n+1$
$T(j, red) \notin tentative.level.m$	disabled after done
Effect:	
$tentative.level.m := tentative.level.m \bigcup \{T(j, red)\}$	
tentative-place-$T(j, yellow)$*-at-level.m*	first check whether token
Precondition:	can be pushed one level down
$T(j, red) \in level.m$	token in level above
$permission_j = granted$	
$\lvert\{k \mid T(k, red) \in level.m\}\rvert < m$	level not full
$T(j, yellow) \notin tentative.level.m$	disabled after done
Effect:	
$tentative.level.m := tentative.level.m \bigcup \{T(j, yellow)\}$	
tentative-place-$T(j, green)$*-at-level.m*	second check whether token
Precondition:	
$T(j, yellow) \in level.m$	
$permission_j = granted$	
$\lvert\{k \mid T(k, red) \in level.m\}\rvert < m$	
$T(j, green) \notin tentative.level.m$	
Effect:	
$tentative.level.m := tentative.level.m \bigcup \{T(j, green)\}$	
assign-X*-to-level.m*	write of the various tokens
Precondition:	
$level.m \subseteq X$	add a new token

$X \subseteq tentative.level.m$ a read enabled the write

Effect:

$level.m := X$

tentative-T_i

Precondition:

$$\forall j \ni T(j, red) \in level.start\ (\forall m(T(j, green) \notin leve.m,\ T(j, red) \in level.m \Rightarrow$$
$$|\{k|T(k, red) \in level.m\}| = m \ or \ permission_j = denied)$$

Effect:

$$\begin{aligned} tentative - T_i = \quad & \{j|\exists m \ni T(j, red) \in level.m, \\ & T(j.yellow) \notin level.m, \\ & |\{k|T(k, red) \in level.m\}| = m\} \end{aligned}$$

assign-T_i

Precondition:

$tentative - T_i \neq \emptyset$

$assigned - T_i = false$

Effect:

$T := T \bigcup \{T_i\}$

$assigned - T_i = true$

assign-S_i

Precondition:

$assigned - T_i = true$

$assigned - S_i = false$

Effect:

$S_i := \bigcap T$

$assigned - S_i = true$

assign-$S_{i,j}$

Precondition:

$assigned - S_i = true$

$j \in S_i$

Effect:

$$\begin{aligned} S_{i,j} := \{k|\exists m, \quad & T(j, red) \notin level.m - 1 \\ & T(j, red) \in level.m \\ & T(k, red) \in level.m\} \end{aligned}$$

assign1-$S_{i,i}$

Precondition:

$i \notin S_i$

$i \in \bigcup T$

Effect:

$S_i := S_i \bigcup \{i\}$

assign2-$S_{i,i}$

Precondition:

$i \notin S_i$

$i \notin \bigcup T$

disabled *private-place-T(i,red)-at-level.m* for all m

Effect:

$S_i := S_i \bigcup \{i\}$

private-place-T(i,red)-at-level.m

Precondition:

$assigned - S_i = true$

$T(i, red) \in level.(m+1)$

$|\{k|T(k, red) \in level.(m+1)\}| < m+1$ or $T(i, yellow) \in level.m+1$

$T(i, red) \notin level.m$

Effect:

$level.m := level.m \bigcup \{T(i, red)\}$

$permission_i := denied$