Transformations for Write-All-With-Collision Model*

Sandeep S. Kulkarni Mahesh Arumugam

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University, East Lansing MI 48824
Email: {sandeep,arumugam}@cse.msu.edu
Web: http://www.cse.msu.edu/~{sandeep,arumugam}

Abstract

Dependable properties such as self-stabilization are crucial requirements in sensor networks. One way to achieve these properties is to utilize the vast literature on distributed systems where such self-stabilizing algorithms have been designed. Since these existing algorithms are designed in read/write model (or variations thereof), they cannot be directly applied in sensor networks. For this reason, we consider a new atomicity model, *write all with collision* (WAC), that captures the computations of sensor networks and focus on transformations from read/write model to WAC model and vice versa. We show that the transformation from WAC model to read/write model is stabilization preserving, and the transformation from read/write model to WAC model is stabilization preserving for timed systems. In the transformation from read/write model to WAC model, if the system is untimed (asynchronous) and processes are deterministic then under reasonable assumptions, we show that (1) the resulting program in WAC model can allow at most one process to execute, and (2) the resulting program in WAC model cannot be stabilizing.

Keywords: dependability, self-stabilization, write-all-with-collision model, read/write model, model conversions

*A preliminary version of this paper appears in [1].

Addr: 3115 Engineering Building, Michigan State University, East Lansing, MI 48824.
Tel:+1-517-355-2387; Fax: +1-517-432-1061;

This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF Equipment Grant EIA-0130724, and a grant from Michigan State University

1 Introduction

Sensor networks are employed in applications such as habitat monitoring [2], active monitoring of intruders [3], and unattended hazard detection [4, 5]. In these applications, sensors are typically deployed in large numbers and in hostile environments. Hence, one of the important requirements in sensor network applications is dependability. One such dependability property is *self-stabilization*. A system is self-stabilizing, if starting from arbitrary initial states, it recovers (in finite time) to state(s) from where the computation proceeds in accordance with its specification [6, 7]. Since sensors are deployed in large volumes and in inaccessible fields, the sensor network should be able to self-stabilize in presence of faults (e.g., message corruption, message/event losses, synchronization errors, etc) without any external intervention. Thus, self-stabilization helps in making the system more robust, available, and maintainable.

Many self-stabilizing algorithms are proposed for distributed systems and networks (for example, leader election [8], communication protocols [9], clock-synchronization [10, 11], consensus [12], spanning tree construction [13, 14]). It is desirable to use these algorithms in the context of sensor networks. However, the model used by these algorithms is not suitable for the constraints (and opportunities) provided by the sensor network. Hence, to utilize these programs, we must transform them so that they can be executed in a model consistent with sensor networks. Moreover, the transformation must preserve the dependability property of interest.

With this motivation, in this paper, we are interested in a new model of computation that occurs in sensor networks. One of the important issues in these networks is message collision: Due to the shared medium, if a sensor simultaneously receives two or more messages then they collide and, hence, the messages become incomprehensible. However, if a message communication does not suffer from a collision then the message is written in the memory of all neighbors of the sender.

Based on the above description, we can view the model of computation in sensor networks as a *write all with collision* (WAC) model (introduced in [15] as a local broadcast model). Intuitively, in this model, in one atomic action, a sensor (process) can update its own state and the state of all its neighbors. However, if two sensors (processes) simultaneously try to update the state of a sensor (process), say k, then the state of k is unchanged. Moreover, in sensor networks, detecting such collisions is difficult due to several reasons. For example, it is possible that a sensor succeeds in updating the state of one of its neighbors even though its update causes collision at another neighbor. Hence, in this paper, we assume that collisions are not detectable.

While previous literature has focused on transformations among other models of computation (e.g., [16–21]), the issue of transformation from (respectively, to) WAC model to (respectively, from) other models has not been considered. And, these transformations are required if we were to use the existing algorithms in the context of sensor networks. To redress this deficiency, we focus on the problem of identifying stabilization preserving transformations that will allow us to transform programs from existing models considered in the literature to WAC model and vice versa, while preserving the self-stabilization property of the original programs. With this intuition, we first discuss some of the commonly used models considered in the literature. Then, we discuss the various transformations that we identify.

Existing models and semantics of distributed programs. Some of the commonly encountered models of computations include a message passing model, a read/write model, and a shared memory model. In these models, a program consists of a set of processes and each process consists of a set of *actions*. And, the tasks that are performed in an action depend upon the model of com-

putation. In message passing model, processes share no memory and they communicate by sending and receiving messages. Thus, in each action, a process can perform one of the following tasks: send a message, receive a message, or perform some internal computation. A read/write model reduces the complexity in modeling message-passing programs. Intuitively, in read/write model, the variables of a process are split into public variables and private variables. In each action, the process can either (1) read the state of one of its neighbors (and update its private variables), or (2) write its own variables (public and private) using its own variables (public and private). (For precise definition of read/write model, we refer the reader to Section 2.) Thus, read/write model allows one to hide the complexities of message queues, message delays, etc. The shared memory model simplifies the read/write model further in that in one action it allows a process to atomically read its state as well as the state of its neighbors and write its own state. Thus, the shared memory model hides the intermediate states, where a process has read the state of a subset of its neighbors that occur in read/write model.

For shared memory model (and also for the WAC model considered in this paper), there are different types of semantics that are often used. Some of the commonly encountered semantics include, interleaving (also known as central daemon), maximum-parallelism and power-set semantics (also known as distributed daemon). In interleaving semantics, given a set of enabled actions, i.e., actions whose execution will change the state of some process, one of those actions is non-deterministically chosen for execution. In maximum-parallelism, all enabled actions (at most one from each process) are executed concurrently. And, in power-set semantics, any non-empty subset of enabled actions (at most one from each process) is executed concurrently.

Contributions of the paper. In this paper, we focus on transformation from read/write model into WAC model under power-set semantics. We show that previously studied concepts such as graph coloring (e.g., [22–24]), local mutual exclusion (e.g., [19, 20]) and time division multiple access (TDMA) [25, 26] can be effectively used for obtaining these transformations. The main contributions of the paper are as follows:

- We present the transformation of programs in read/write model into programs in WAC model for untimed (asynchronous) systems. In this transformation, we show that if the transformed program is deterministic and cannot perform redundant writes (cf. Section 3 for definition) then at most one process can execute at a time. Also, we argue that a self-stabilization preserving transformation is not possible in this model.

- For timed systems, we present the transformation of programs in read/write model into programs in WAC model. This transformation allows concurrent executions of multiple processes. We show that if the given program is self-stabilizing then the transformed program is also self-stabilizing for a fixed topology. In other words, the transformation is stabilization preserving.

- We illustrate our transformation algorithms using the logical grid routing program [27, 28] designed for the Line in the Sand project [3]. We transform the routing program in read/write model into a program in WAC model and show that the transformed program is similar to the implementation in [3].

- We present the transformation of programs in WAC model into programs in read/write model. This transformation does not assume that the topology is fixed or known in advance. We show that this transformation is also stabilization preserving.

Organization of the paper. The rest of the paper is organized as follows. In Section 2, we introduce the read/write model and the WAC model. In addition, we state the assumptions made in this paper. In Section 3, we present the transformation of programs in read/write model into programs in WAC model under power-set semantics for untimed systems. Then, in Section 4, we present the transformation for timed systems. Subsequently, in Section 5, we illustrate our transformation algorithms using the grid routing protocol designed in [27, 28]. In Section 6, we present the transformation of programs in WAC model under power-set semantics into programs in read/write model. Finally, in Section 7, we discuss some of the questions raised by this work and in Section 8, we summarize the results and identify future research directions.

2 Preliminaries

In this section, we first precisely specify the structure of the programs written in read/write model and in WAC model. Then, we present the assumptions made about the underlying system.

The programs are specified in terms of guarded commands; each guarded command (respectively, action) is of the form:

$$guard \quad \longrightarrow \quad statement,$$

where *guard* is a predicate over program variables, and *statement* updates program variables. An action $g \longrightarrow st$ is enabled when g evaluates to true and to execute that action, st is executed. A computation of this program consists of a sequence $s_0, s_1, \ldots$, where s_{j+1} is obtained from s_j by executing actions (one or more, depending upon the semantics being used) in the program.

A computation model limits the variables that an action can read and write. Towards this end, we split the program actions into a set of processes. Each action is associated with one of the processes in the program. We now describe how we model the restrictions imposed by the read/write model and the WAC model.

Read/Write model. In read/write model, a process consists of a set of public variables and a set of private variables. In the read action, a process reads (one or more) public variables of one of its neighbors. For simplicity, we assume that each process j has only one public variable $v.j$ that captures the values of all variables that any neighbor of j can read.

Furthermore, in a read action, a process could read the public variables of its neighbor and write a different value in its private variable. For example, consider a case where each process has a variable x and j wants to compute the sum of the x values of its neighbors. In this case, j could read the x values of its neighbors in sequence. Whenever j reads $x.k$, it can update a private variable $sum.j$ to be $sum.j + x.k$. Once again, for simplicity, we assume that in the read action where process j reads the state of k, j simply copies the public variables of k. In other words, in the above case, we require j to copy the x values of all its neighbors and then use them to compute the sum.

Based on the above discussion, we assume that each process j has one public variable, $v.j$. It also maintains $copy.j.k$ for each neighbor k of j; $copy.j.k$ captures the value of $v.k$ when j read it last. Now, a read action by which process j reads the state of k is represented as follows:

$$true \quad \longrightarrow \quad copy.j.k = v.k$$

And, the write action at j uses $v.j$ and $copy.j$ (i.e., copy variables for each neighbor) and any other private variables that j maintains to update $v.j$. Thus, the write action at j is as follows:

$$predicate(v.j, copy.j, other\_private\_variables.j) \longrightarrow \text{update } v.j,\ other\_private\_variables.j;$$

WAC model. In the WAC model, each process consists of write actions (to be precise, write-all actions). Each write action at j writes the state of j and the state of its neighbors. Similar to the case in read/write model, we assume that each process j has a variable $v.j$ that captures all the variables that j can potentially write to any of its neighbors. Likewise, process j maintains $l.j.k$ for each neighbor k; $l.j.k$ denotes the value of $v.k$ when k wrote it last. Thus, an action in WAC model is as follows:

$$predicate(v.j, l.j, other\_private\_variables.j) \longrightarrow \text{update } v.j,\ other\_private\_variables.j;\ \forall k : k \text{ is a neighbor of } j : l.k.j = v.j;$$

We model the collision as follows. If two neighbors, say j and k, of process q were to execute simultaneously then their write actions collide at q. Hence, neither $l.q.j$ nor $l.q.k$ is updated. Thus, if several processes execute simultaneously then only a subset of their neighbors may be updated.

We refer the reader to Section 5 for an example of a program written in read/write model and in WAC model.

Remark. In the rest of the paper, we leave the additional private variables considered above implicit.

Preserving stabilization. We now discuss our approach to show that a transformation algorithm preserves stabilization. Towards this end, we define the notion of *equivalence* between a computation of the given program and computation of the transformed program. This notion is based on the definition of refinement [29, 30] and simulation [30].

Consider the transformation of program p in read/write model into program p' in WAC model. Note that in WAC model, multiple writes can occur at once whereas in read/write model, at most one read/write can occur at a time. Hence, each step of the program in WAC model would be simulated in read/write model by multiple steps. Hence, if $c' = \langle s_0, s_1, \ldots \rangle$ is a computation of p' and c is a computation of p, we say that c and c' are equivalent if c is of the form $\langle t_{00}, t_{01}, \ldots, t_{0f_0} (= t_{10}), t_{11}, \ldots, t_{1f_1} (= t_{20}), \ldots \rangle$, where $\forall j : s_j$ and t_{j0} are identical (subject to renaming of variables) as far as the variables of p are concerned. For the transformation of p' in WAC model into p in read/write model, the definition of equivalence is similar. We require that c can be expressed as $\langle t_{00}, t_{01}, \ldots \rangle$ by commuting the read/write actions in c in such a way that the effect of these transitions does not change due to such commutation.

To show that our transformations are stabilization preserving, we proceed as follows: Let p be the given program, say, in read/write model, and let p' be the transformed program in WAC model. We show that given any computation of p', there exists a suffix of that computation such that there is a computation of p that is equivalent to that suffix. If the given program, p, is stabilizing fault-tolerant then any computation of p is guaranteed to reach legitimate states and satisfy the specification after reaching legitimate states. It follows that eventually, a computation of p' will reach legitimate states from where it satisfies its specification.

System assumptions. We assume that the set of processes in the system are connected. If the set of processes are partitioned then the algorithms in this paper can be executed for each partition. Further, we assume that in the given program in read/write model, for any pair of neighbors j and k, j can never conclude that k does not need to read the state of k. In other words, we require that the transformation should be correct even if each process executes infinitely often. We also assume that the transformation should work correctly even if the communication among neighbors is bidirectional, i.e., the transformation algorithm should not assume that communication between some neighbors be only unidirectional. Further, in our transformation from read/write model to WAC model, we assume that the topology remains fixed during the program execution, i.e., failure or repair of processes does not occur. Thus, while proving stabilization, we disallow corruption of topology related information. This assumption is similar to assumptions in previous stabilizing algorithms where the process IDs are considered to be incorruptible. We note that our transformation from WAC model to read/write model does not assume that the topology is known up front and the topology can change at run time.

We consider two types of systems, timed and untimed. In a timed system, each process has a clock variable. We assume that the rate of increase of clock is same for all the processes. We assume that there exists a distinguished process (or a base station) in the network that is responsible for initiating the computation of the program in WAC model for timed systems. In an untimed (asynchronous) system, processes do not have the notion of time or the speed of processes. In both these systems, we assume that the execution of an action is instantaneous. The time is consumed *between* successive actions. Additionally, in an untimed system, we assume that the delay between successive actions is unpredictable.

3 Read/Write Model to WAC Model in Untimed Systems

In this section, we discuss the first of our transformations where we transform a program in read/write model into a program in WAC model. In the WAC model, there is no equivalent of a read action. Hence, an action by which process j reads the state of k in the read/write model needs to be modeled in the WAC model by requiring process k to write the appropriate value at process j. Of course, when k writes the state of j in this manner, it is necessary that no other neighbor of j is writing the state of j at the same time. Finally, a write action in read/write model can be executed in WAC model as is.

To obtain the transformed program that is correct in WAC model, we organize the processes in the given program (in read/write model) in a ring. Such a ring can be *statically* embedded in any arbitrary graph by first embedding a spanning tree in it and then using an appropriate traversal mechanism to ensure that each process appears at least once in the ring (cf. Figure 1).

Let the processes in this ring be numbered $0 \ldots n{-}1$; note that if a process from the original graph is repeated in the ring then that process gets multiple numbers in this ring. Now, in the transformed program, process 0 executes first. When process 0 executes and writes the state of process 1 (and any other processes that are neighbors of process 0 in the original communication graph), process 1 is enabled and permitted to execute. When process 1 executes, it allows process 2 to execute, and so on. Figure 2 shows the actions of process j in the transformed program. If a process in the original graph has multiple numbers in the ring, it executes the actions corresponding to all those values.

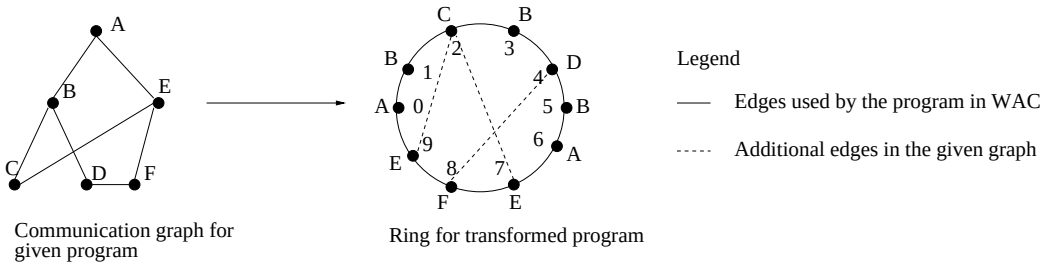

Figure 1: In the transformed program, 'A' will execute actions of processes 0 and 6, 'B' will execute actions of processes 1, 3 and 5, and so on.

process j
const n; // number of processes in the ring
var $v.j$, $l.j$, $counter.j$;
initially
set $v.j$ according to the initial value of $v.j$ in read/write model;
set $l.j$ according to the initial value of $copy.j$ in read/write model;
$counter.j = 0$;
begin
$counter.j{=}j \longrightarrow$ if($predicate(v.j, l.j)$) update $v.j$;
$counter.j{=}(j+1)\ mod\ \ n$;
$\forall k : k$ is a neighbor of $j : l.k.j, counter.k = v.j, (j+1) mod\ n$;
// this write action will enable process numbered $j+1$
end

Figure 2: Read/write model to WAC model

Theorem 3.1 For every computation of the transformed program in WAC model under power-set semantics there is an equivalent computation of the given program in read/write model.

Proof. The main idea in this proof is that every step of j in the transformed program is equivalent to the following computation of the original program in read/write model: (write action by j, followed by read action by each neighbor of j).

Consider a computation $s_0, s_1, s_2, \ldots$ of the transformed program in WAC model. Based on the initial values of the counters, (s_0, s_1) is a transition of process 0. Let $x_1, x_2, \ldots, x_{f_0-1}$ be the neighbors of 0. Now, for the transition (s_0, s_1), we construct a computation $\langle t_{00}, t_{01}, t_{02}, \ldots, t_{0f_0}\rangle$ of the original program. Since t_{ab} is a state of the program in the read/write model, we identify the v and $copy$ values for t_{ab}.

- For state t_{00}: $\forall j :: v.j(t_{00}) = v.j(s_0)$, $\forall j, k :: copy.j.k(t_{00}) = l.j.k(s_0)$.
- State t_{01} is obtained from t_{00} by executing the write action at process 0 (i.e., the process that was numbered 0 during transformation).
- State $t_{0w}, 1 < w \leq f_0$, is obtained from $t_{0(w-1)}$ where process x_{w-1} reads the value of $v.0$.

Now, in $t_{10} = t_{0f_0}$, we have $\forall j :: v.j(t_{10}) = v.j(s_1)$, $\forall j, k :: copy.j.k(t_{10}) = l.j.k(s_1)$. Further, by

induction, if $s_0, s_1, s_2, \ldots$ is a computation of the transformed program in WAC model then there exists an equivalent computation $t_{00}, t_{01}, \ldots, t_{0f_0}(= t_{10}), t_{11}, \ldots$ that is a computation of the given program in read/write model. □

3.1 Optimality Issues

Now, we discuss the optimality of the transformation algorithm presented in Figure 2. Towards this end, we first identify the notion of redundant writes. Based on the definition of WAC model, the guard of any action, say of process j, depends only on the local variables of j. Now, consider the case where process j executes its action (and writes its state as well as the state of its neighbors). If an action of j is still enabled after this execution, it follows that j can again execute and write the state of its neighbors. In this scenario, one of the writes of j is redundant if the system was untimed and no collision had occurred. To show the optimality of the above transformation, we assume that processes do not perform such redundant writes.

Redundant writes. We say that process j does not perform redundant writes if after the execution of any action by j, all actions of j are disabled until a neighbor of j executes and writes the state of j.

Theorem 3.2 In an untimed system where processes cannot detect collisions, any algorithm that (1) transforms a given program in read/write model into an equivalent program in WAC model under power-set semantics, and (2) ensures that the processes in the generated program in WAC model are deterministic and do not perform redundant writes, must produce programs in WAC model in which at most one process executes at a time.

Proof. We present the proof of this theorem in Appendix A. A reader should note that while the statement of this theorem is important for the discussion of subsequent results, the subsequent results do not require the knowledge of the proof. □

There are several ways to improve the performance of the transformation if we weaken some of the assumptions made above. Specifically, we can remove the assumption that processes cannot perform redundant writes in order to allow concurrency in the programs in WAC model (cf. Appendix B). Alternatively, we can remove the assumption that the processes are deterministic. In [15], the authors present the transformation algorithms where the processes are randomized. Another approach would be to remove the assumption about untimed systems. If the processes are allowed to use time, we can design transformation algorithms that allow more concurrency. In Section 4, we present such algorithms. Finally, we can weaken the assumption that the processes cannot detect collisions. However, we are not aware of any transformations where collisions are detectable.

3.2 Impossibility of Preserving Stabilization in Untimed Systems

Now, under the assumptions stated in the above theorem, we show that any transformation algorithm cannot be stabilization preserving. Based on the assumption that processes do not perform redundant writes, for each process, there exists a local state of that process where none of its actions are enabled. Also, since the guard of an action at a process depends only on its local variables, we can perturb the given program to states where none of the actions are enabled. It follows that it will not be possible for the program to reach legitimate states from such a state.

In the context of the above result, a reader may wonder if a stabilizing token ring circulation algo-

rithm (e.g., [31]) could be used to achieve stabilization. To add stabilization to the transformation, we need a stabilizing token ring circulation algorithm that is correct under the WAC model. By contrast, existing stabilizing token ring circulation algorithms are correct under read/write model.

The above impossibility result depends on the assumption that the system is untimed and processes are deterministic, cannot detect collisions, and cannot perform redundant writes. If the assumption about untimed system is removed then it is possible to preserve stabilizing fault-tolerance. We present such stabilization preserving transformations in Section 4.

4 Read/Write Model to WAC Model in Timed Systems

In this section, we present transformations for a program in read/write model into a program in WAC model for timed systems. Specifically, we present transformations, first, for a grid topology and then for an arbitrary topology using graph coloring. These transformations can be achieved using a collision-free time-slot based protocol like time-division multiple access (TDMA) [25, 26]. In these transformations, we initially assume that the clocks of the processes are initialized to 0 and the rate of increase of the clocks is same for all processes. Subsequently, we also present an approach to deal with uninitialized clocks; this approach enables us to ensure that if the given program in read/write model is stabilizing fault-tolerant then the transformed program in WAC model is also stabilizing fault-tolerant.

4.1 Transformation Algorithm For a Grid Topology

This algorithm is based on the TDMA algorithm presented in [25]. Let the processes be arranged in a 2 dimensional grid topology as shown in Figure 3. Also, assume that there is a distinguished process (base station) at the left-top position (process $\langle 0,0\rangle$). This distinguished process starts the computation in the transformed program. First, this distinguished process executes and writes the state of processes $\langle 1,0\rangle$ and $\langle 0,1\rangle$. Now, either $\langle 1,0\rangle$, $\langle 0,1\rangle$ or both can execute. However, if both $\langle 1,0\rangle$ and $\langle 0,1\rangle$ execute simultaneously, their write actions will collide at process $\langle 1,1\rangle$. Hence, we use the following algorithm to ensure collision-freedom during write actions: process $\langle 1,0\rangle$ executes one slot (i.e., time required to execute one action in the WAC model) after $\langle 0,0\rangle$ writes its state. And, process $\langle 0,1\rangle$ executes after two slots. In general, if the distinguished process starts the computation when its *clock* is equal to 0, then process $\langle a,b\rangle$ executes at $clock = a + 2b$.

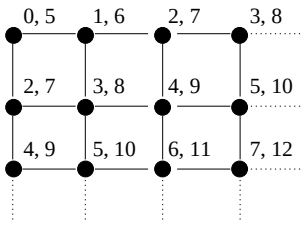

Figure 3: Time slots for processes in a grid. The numbers associated with a process show the first two slots in which it could execute.

Further, in the above algorithm, the distinguished process can execute again when its clock equals 5 (cf. Figure 3). In this case, the write action of $\langle 0,0\rangle$ does not collide with simultaneous write

actions of other processes. Hence, in general, process j located at $\langle a, b\rangle$ can execute and write its state to its neighbors at $slot.j + c * 5$, where $slot.j (= a + 2b)$ is the initial slot assignment and c is an integer. Figure 4 shows the actions of the process j at location $\langle a, b\rangle$.

```
process j                          // located at ⟨a, b⟩
const  slot.j = a + 2b;            // by symmetry, slot.j could also be initialized to 2a + b
       period = 5;
var    v.j, l.j, clock.j;
initially
       set v.j according to the initial value of v.j in read/write model;
       set l.j according to the initial value of copy.j in read/write model;
       clock.j = 0;
begin
       (∃c : clock.j = slot.j + c * period) ⟶ if(predicate(v.j, l.j)) update v.j;
                                              ∀k : k is a neighbor j : l.k.j = v.j;
end
```

Figure 4: Transformation for grid topology

Theorem 4.1 For every computation of the transformed program in WAC model under power-set semantics there is an equivalent computation of the given program in read/write model.

Proof. Consider the program in the WAC model (cf. Figure 4). Based on the initial values of the clocks, the distinguished process (process $\langle 0, 0\rangle$) starts the computation. Further, the initial values of the variables of the program are assigned according to the program in the read/write model.

Similar to the proof of Theorem 3.1, a write action by process j in the transformed program is equivalent to the following computation of the original program in read/write model: a write by process j, followed by the read actions by the neighbors of j. Thus, if only one process executed at every time slot in WAC model then, from Theorem 3.1, it follows that for every computation of the transformed program in WAC model there is a corresponding computation in read/write model.

Now, consider the case where multiple processes simultaneously execute in WAC model. Based on the slot assignment, there are no collisions and, hence, these multiple write-all actions can be serialized. If $\langle s_0, s_1\rangle$ is a transition of the transformed program in WAC model under power-set semantics then there is a corresponding computation $\langle u_0, u_1, \ldots, u_m\rangle$ of the program in interleaving semantics, where m is the number of processes that execute simultaneously in the transition $\langle s_0, s_1\rangle$. Moreover, for each transition $\langle u_j, u_{j+1}\rangle$ in interleaving semantics, there is a corresponding computation of the program in read/write model (cf. Theorem 3.1). Thus, for each transition of the transformed program in WAC model under power-set semantics, there is a sequence of transitions of the transformed program in read/write model. It follows that, for a computation of the transformed program in the WAC model under power-set semantics, there is an equivalent computation of the given program in the read/write model. □

Observation 4.2 There exists a suffix of the computation of the transformed program where the number of processes enabled at any instant of time is maximal.

Proof. From the slot assignment algorithm, it follows that the number of processes that execute at a time is maximal [25]. Thus, there exists a suffix of the computation of the transformed program in WAC model where the number of processes enabled simultaneously is maximal. □

4.2 Transformation Algorithm For an Arbitrary Topology

The main idea behind this algorithm is graph coloring. Let the communication graph in the given program in read/write model be $G=(V,E)$. We transform this graph into $G'=(V,E')$ such that $E'=\{(x,y)|(x\neq y)\wedge((x,y)\in E\vee(\exists z::(x,z)\in E\wedge(z,y)\in E))\}$ (cf. Figure 5). In other words, two distinct vertices x and y are connected in G' if the distance between x and y in G is at most 2. Let $f:V\rightarrow[0\ldots K-1]$ be the color assignments such that $(\forall j,k:k$ is a neighbor of j in $G':f(j)\neq f(k))$, where K is any number that is sufficient for coloring G'.

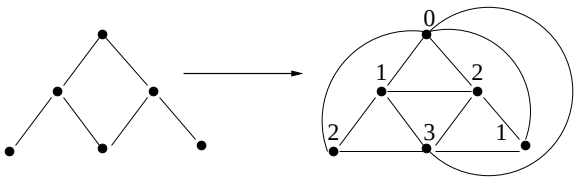

Figure 5: Transformation using graph coloring. The number associated with each process denotes the color of the process.

Let $f.j$ denote the color of process j. Now, process j can execute at $clock.j=f.j$. Moreover, process j can execute at time slots $f.j+c*K$, where c is an integer. When j executes, it writes its own state and the state of its neighbors in G. Based on the color assignment, it follows that two write actions do not collide. Figure 6 shows the actions of process j.

process j
const $f.j$, K;
var $v.j$, $l.j$, $clock.j$;
initially
set $v.j$ according to the initial value of $v.j$ in read/write model;
set $l.j$ according to the initial value of $copy.j$ in read/write model;
$clock.j=0$;
begin
$(\exists c:clock.j=f.j+c*K)\longrightarrow$ if($predicate(v.j,l.j)$) update $v.j$;
$\forall k:k$ is a neighbor of j in the original graph :
$l.k.j=v.j$;
end

Figure 6: Transformation for arbitrary topology

Theorem 4.3 For every computation of the transformed program in WAC model under power-set semantics there is an equivalent computation of the given program in read/write model.

Proof. The proof of this theorem is identical to that of Theorem 4.1; the only difference in this algorithm is the color assignment to ensure collision freedom. Once collision freedom is achieved, the resulting proof remains the same. □

Theorem 4.4 If the maximum degree of G is d, then the period between successive executions of a process is at most d^2+1.

Proof. If the maximum degree of G is d then the maximum degree in G' would be at most d^2.

In [24], it has been shown that if each subgraph of a graph has a vertex of degree $K-1$ or less, then the graph can be colored with K colors. Hence, G' can be colored with at most $K = d^2 + 1$ colors. Thus, the period between successive executions of process j is at most $d^2 + 1$. □

Remark. The above theorem presents the upper bound on the performance of the transformation. For sensor networks, maximum degree d is usually small. Hence, the period between successive executions of a process is small. Moreover, $d^2 + 1$ is the upper bound for the period and the period can be smaller than $d^2 + 1$. For example, for a grid topology, maximum degree, $d = 4$, and $period = K = 5$ (cf. Section 4.1).

4.3 Preserving Stabilization in Timed Systems

To show that stabilization is preserved during transformation, we first present how we deal with the case where the clocks are not initialized or clocks of processes are corrupted. Note that we have assumed that the rate of increase of the clocks is still the same for all processes. To recover from uninitialized clocks, we proceed as follows: Initially, we construct a spanning tree of processes rooted at the base station. Let $p.j$ denote the parent of process j in this spanning tree. Process j is initialized with a constant $c.j$ which captures the difference between the initial slot assignment of j and $p.j$.

If clocks are not synchronized, the action by which $p.j$ writes the state of j may collide with other write actions in the system. Process j uses the absence of this write to stop and wait for synchronizing its clock. Process j waits for $K * n$ slots, where K is the period between successive slots and n is the number of processes in the system. This ensures that process j starts executing only when all its descendants have stopped in order to synchronize their clocks. When j later observes the write action performed by $p.j$, it can use $c.j$ to determine the next slot in which it should execute. Since the root process or the base station continues to execute in the slots assigned to it, eventually, its children will synchronize with the root. Subsequently, the grandchildren of the root will synchronize, and so on. Continuing thus, the clocks of all processes will be synchronized and hence, further computation will be collision-free.

In the absence of topology changes, the spanning tree constructed above and the values of $f.j$ and $c.j$ are constants. Hence, for a fixed topology, we assume that these values are not corrupted. (This assumption is similar to assumptions elsewhere where process ID cannot be corrupted.) Once the clocks are synchronized, from Theorem 4.3, for the subsequent computation of the transformed program, there is an equivalent computation of the given program in read/write model. Also, if the given program is stabilizing fault-tolerant then every computation of that program reaches legitimate states. Combining these two results, it follows that every computation of the transformed program eventually reaches legitimate states. Thus, we have

Theorem 4.5 If the given program in read/write model is stabilizing fault-tolerant then transformed program (with the modifications discussed above) in WAC model is also stabilizing fault-tolerant. □

Remark. The above modification is meant to illustrate the feasibility of preserving stabilization while transforming a program in read/write model into a program in WAC model. The time for which a process waits, $K * n$ slots, after failing to observe the write action of its parent is an overestimate. The optimal time for which a process should wait is outside the scope of the paper.

5 Illustration: Transformation of a Routing Program

In this section, we illustrate the transformation algorithm from read/write model into WAC model. Specifically, we consider the logical grid routing protocol (LGRP) [27, 28] designed for A Line in the Sand (LITeS) experimental project [3], where a sensor network is used for target detection, classification and tracking. We note that the LGRP program is a variation of the balanced routing program proposed in [32], modified to work in the context of grid based network topology. We transform the LGRP program in read/write model into a program in WAC model. The resulting program is a variation of the program used in LITeS project [3]. (There are small differences between the transformed program and the program in [3]. We identify them at the end of this section.)

In LGRP, sensors are organized into a logical grid, either offline or using a localization algorithm. A routing tree is dynamically constructed with the base station as the root. The base station is located at $\langle 0, 0\rangle$ of the logical grid. A sensor (say, j) classifies its neighbors within H hops as low neighbors or high neighbors. Specifically, sensor k, located within H hops of j, is classified as j's low neighbor if k's distance to the base station in the logical grid is less than that of j's distance. Otherwise, it is classified as j's high neighbor. The set of low neighbors and high neighbors identify the potential parents of j in the routing tree. The low and high neighbors are computed statically (or at initialization) and are assumed to be constant as far as LGRP is concerned.

Sensor j maintains a variable, *inversion count*. The inversion count of the base station is 0. If j chooses one of its low neighbors as its parent then it sets its inversion count to that of its parent. Otherwise, j sets its inversion count to inversion count of its parent + 1. If j finds a neighbor which gives a smaller inversion count, then it replaces its parent and updates its inversion count. Furthermore, when the inversion count exceeds a certain threshold, it indicates that the tree may be corrupted (i.e., contain cycles). To deal with this problem, when the inversion count exceeds a predetermined threshold, j sets its parent to *null*. Sensor j will later rejoin the routing tree when it finds a neighbor which provides a better inversion count. Once the routing tree is established, whenever j receives a data message, it forwards the message to its parent.

Next, we specify the program in read/write model and then transform it into WAC model. In this illustration, we consider $H = 1$, to simplify the presentation of the program. We note that our transformations can be used for other values as well.

LGRP program in read/write model. Figure 7 shows the LGRP program in read/write model. In this program, process j $(1..N-1)$ maintains 2 public variables, $inv.j$, inversion count of j, and $up.j$, status of j (indicates whether j has failed or not).[1] Also, process j maintains a private variable, $p.j$, the parent of j and 2 copy variables, $copy.j.inv.k$, the inversion count values of neighbors of j when j read last, and $copy.j.up.k$, the status of neighbors of j. The action of the base station (i.e., process bs) is as follows:

$$true \longrightarrow p.bs, inv.bs, up.bs = bs, 0, true;$$

In this program, process j reads the state of its neighbors and updates its copy variable, $copy.j.inv.k$ and $copy.j.up.k$. When j finds a low neighbor or a high neighbor that gives a better inversion count

[1] In this program, whenever a sensor/process fails, it notifies its neighbors. This action can be implemented as follows; whenever a sensor fails to read its neighbor for a threshold number of consecutive attempts, it declares that neighbor as failed. Similarly, in WAC model, whenever a sensor fails to receive update from its neighbor, it declares that neighbor as failed.

value, it replaces its parent and updates the inversion count accordingly. If $p.j$ has failed, inversion count of $p.j$ has reached the maximum allowed value, or inversion count of j is not consistent with its parent, j removes itself from the routing tree by setting $p.j$ to $null$ and $inv.j$ to $cmax$. It will rejoin the routing tree when it finds a neighbor that provides a better inversion count. Thus, the LGRP program in read/write model is as shown in Figure 7.

process $j : (1..N-1)$
const
  $cmax$; // maximum inversion count
  $ln.j$; // low neighbors of j
  $hn.j$; // high neighbors of j
var
  public $inv.j : \{0..cmax\}$; // inversion count of j
  public $up.j : \{true, false\}$; // status of j
  private $p.j : \{ln.j \cup hn.j \cup null\}$; // parent of j
  copy $\forall k \in \{ln.j \cup hn.j\} : copy.j.inv.k$; // inversion count of neighbors of j
  copy $\forall k \in \{ln.j \cup hn.j\} : copy.j.up.k$; // status of neighbors of j
begin
  $true \longrightarrow copy.j.inv.k = inv.k$; // read $inv.k$
    $copy.j.up.k = up.k$; // read $up.k$

  $k \in ln.j \wedge copy.j.up.k \wedge (copy.j.inv.k < cmax) \wedge (copy.j.inv.k < inv.j)$
    $\longrightarrow p.j, inv.j = k, copy.j.inv.k$; // low neighbor which gives a better
    // path to the base station (i.e., root)

  $k \in hn.j \wedge copy.j.up.k \wedge (copy.j.inv.k < cmax) \wedge (copy.j.inv.k + 1 < inv.j)$
    $\longrightarrow p.j, inv.j = k, copy.j.inv.k + 1$; // high neighbor which gives a better
    // path to the base station (i.e., root)

  $p.j \neq null \wedge$
    $(\neg copy.j.up.(p.j) \vee$ // parent is dead, or
    $(copy.j.inv.(p.j) >= cmax) \vee$ // parent's inversion count exceeds $cmax$, or
    $(p.j \in ln.j \wedge inv.j \neq copy.j.inv.(p.j)) \vee$ // j's inversion count is not
    $(p.j \in hn.j \wedge inv.j \neq copy.j.inv.(p.j) + 1))$ // consistent with its parent
    $\longrightarrow p.j, inv.j = null, cmax$;

  $p.j = null \wedge inv.j < cmax$ // j has no parent and its
    $\longrightarrow inv.j = cmax$; // inversion count is less than $cmax$
end

Figure 7: Grid routing program in read/write model

LGRP program in WAC model. Figure 8 shows the transformed routing program in WAC model. We obtained this program using the transformation algorithm proposed in Section 4.2. In this program, process j $(1..N-1)$ knows the color assigned to it $(f.j)$ and the maximum number of colors used in the system (K). Furthermore, j maintains 2 copy variables, $l.j.inv.k$ and $l.j.up.k$ similar to the copy variables in the program in read/write model. However, $l.j.inv.k$ and $l.j.inv.k$ indicates the value of $inv.k$ and $up.k$ respectively when process k last wrote j. The action of the base station (i.e., process bs) is as follows, where $f.bs$ is the color assigned to the base station:

$\exists c : clock.bs = f.bs + c * K \longrightarrow p.bs, inv.bs, up.bs = bs, 0, true;$
$\forall k : k$ is a neighbor of $bs : l.k.inv.bs, l.k.up.bs = inv.bs, up.bs;$

Also, in the time slot assigned to j, process j executes the write actions consistent with the write actions in the program in read/write model. Specifically, j replaces its parent and updates its inversion count if it finds a neighbor that gives a better inversion count value. Also, it sets $p.j$ to $null$ and $inv.j$ to $cmax$ if its current parent has failed, inversion count of its current parent exceeds the threshold, or j's inversion count is not consistent with that of its current parent. Once j updates its local variables, it executes its write-all action. In other words, j updates $inv.j$ and $up.j$ at its neighbors.

process $j : (1..N-1)$
const
  $cmax, ln.j, hn.j;$
  $f.j;$ // color of j
  $K;$ // maximum colors used in the network
var
  public $inv.j$, $up.j$;
  private $p.j$, $clock.j$ (initially, $clock.j = 0$);
  copy $\forall k \in \{ln.j \cup hn.j\} : l.j.inv.k;$ // inversion count of neighbors of j
  copy $\forall k \in \{ln.j \cup hn.j\} : l.j.up.k;$ // status of neighbors of j
begin
  $(\exists c : clock.j = f.j + c * K)$
    $\longrightarrow$
    // execute write action consistent with the write action in the read/write program
    if $(k \in ln.j \wedge l.j.up.k \wedge (l.j.inv.k < cmax) \wedge (l.j.inv.k < inv.j))$
      $p.j, inv.j = k, l.j.inv.k;$

    if $(k \in hn.j \wedge l.j.up.k \wedge (l.j.inv.k < cmax) \wedge (l.j.inv.k + 1 < inv.j))$
      $p.j, inv.j = k, l.j.inv.k + 1;$

    if $(p.j \neq null \wedge$
      $(\neg l.j.up.(p.j) \vee$
      $(l.j.inv.(p.j) >= cmax) \vee$
      $(p.j \in ln.j \wedge inv.j \neq l.j.inv.(p.j)) \vee$
      $(p.j \in hn.j \wedge inv.j \neq l.j.inv.(p.j) + 1)))$
      $p.j, inv.j = null, cmax;$

    if $(p.j = null \wedge inv.j < cmax)$
      $inv.j = cmax;$

    // execute write-all actions, updating the state of all the neighbors of j
    $\forall k : k \in \{ln.j \cup hn.j\} : l.k.inv.j, l.k.up.j = inv.j, up.j;$
end

Figure 8: Grid routing program in WAC model

Thus, the program written in read/write model is transformed into a program in WAC model. The transformed program shown in Figure 8 differs from the program in [27, 28]. First, unlike the program in [27, 28], the transformed program uses TDMA to execute the write-all actions.

This ensures collision-free update of the state of a sensor at its neighbors. Next, the transformed program abstracts the failure of sensors. In [27,28], the authors use "heartbeat" messages; the lack of a threshold number of such messages indicates failure of sensors.

6 WAC Model to Read/Write Model

In this section, we focus on the transformation of a program that is correct in the WAC model under power-set semantics into a program in read/write model. During this transformation, an action by which j writes its own state and the state of its neighbors in WAC model is split so that j writes its own state and then allows each of its neighbors to read its state.

However, if multiple fragmented WAC actions are executed simultaneously in the read/write model, their execution may not correspond to the sequential (respectively, parallel) execution of those actions in the WAC model. For example, consider the execution of two actions, $a.j$ and $a.k$, at neighboring processes j and k in the WAC model. In a fragmented execution of these two actions, the following execution scenario is feasible: j writes its own state as prescribed by action $a.j$, k writes its own state as prescribed by $a.k$, j reads the state of k, and k reads the state of j. However, such a scenario is not possible in WAC model. Specifically, if $a.j$ and $a.k$ are executed simultaneously then, due to collision, j (respectively, k) will not be able to write the state of k (respectively, j). And, in a sequential execution where $a.k$ is executed after $a.j$, k will be aware of the new state of j and use that in the execution of $a.k$. By contrast, in the above fragmented execution, this property was not true. Thus, the fragmented execution of two actions in the WAC model may not correspond to their sequential or parallel execution. Hence, to transform the given program in WAC model, we ensure that two neighboring processes do not simultaneously execute their fragmented WAC actions.

The problem of ensuring that neighboring processes do not execute simultaneously is a well-known problem in distributed computing. It has been studied in the context of local mutual exclusion (e.g., [20]), dining philosophers (e.g., [18,33]) and drinking philosophers (e.g., [18,33]). Since either of these solutions suffices for our purpose, we simply describe the features of these solutions that are of importance here.

Each of the solutions in [18, 20, 33] has the following two actions: *enterCS* and *exitCS*. These solutions further guarantee that when a process is in its critical section (i.e., it has executed *enterCS* but not *exitCS*) none of its neighbors are in its critical section. Using these two actions, in Figure 9, we demonstrate how one can transform a program in WAC model into a program in read/write model. Note that the last action in Figure 9 (i.e., action A3) appears to allow j to read the state of all its neighbors; this action can be *slowly* executed so that j reads the counters of its neighbors one at a time.

Theorem 6.1 For every computation of the transformed program in read/write model there is an equivalent computation of the given program in WAC model under power-set semantics.

Proof. Consider a computation of the transformed program in read/write model (cf. Figure 9). In the absence of state perturbations, we have, $(\forall j, k :: counter.k.j \leq counter.j.j)$. Now, when j executes *enterCS*, it increments $counter.j.j$. It follows that j cannot execute *exitCS* until all neighbors of j copy the new value of $v.j$. Based on the guarantees of local mutual exclusion, neighbors of j do not execute until j exits critical section. Thus, the write action at j followed by read action by neighbors of j is equivalent to the action in WAC model that writes the state of j

process j
var $v.j$, $copy.j$, $counter.j$;
initially
set $v.j$ according to the initial value of $v.j$ in WAC model;
set $copy.j$ according to the initial value of $l.j$ in WAC model;
$\forall k :: counter.j.k = 0$;
begin

A1: upon executing $enterCS$	$\longrightarrow$	$counter.j.j := counter.j.j + 1$; execute 'write part' of the program in WAC model to update $v.j$;
A2: $counter.j.k < counter.k.k$	$\longrightarrow$	$counter.j.k, copy.j.k := counter.k.k, v.k$;
A3: $(\forall k : counter.k.j \geq counter.j.j)$	$\longrightarrow$	execute $exitCS$; request for CS again;

end

Figure 9: WAC model to read/write model

and all its neighbors.

Further, in the transformation shown in Figure 9, it is guaranteed that two neighbors are not in the middle of executing a write action. Thus, for the computation of the transformed program in read/write model, there is an equivalent computation where the corresponding actions in WAC model are executed sequentially. In other words, for every computation of the transformed program in read/write model there is an equivalent computation of the original program in WAC model under interleaving semantics. Based on the definition of power-set semantics, for every computation of the transformed program in read/write model there is an equivalent computation of the original program in WAC model under power-set semantics. □

6.1 Preserving Stabilization

Now, we show that if we use a local mutual exclusion algorithm that is stabilizing (e.g., [18, 20]) then the transformation shown in Figure 9 is stabilization preserving. To show this, we first observe that any stabilizing solution for local mutual exclusion must ensure that eventually some process enters its critical section. Consider the case where process j is in critical section. If there exists a neighbor, say k, of j such that $counter.k.j < counter.j.j$ then k will copy $counter.j.j$. Thus, eventually, j will be able to exit critical section. Based on the above discussion, it follows that the stabilization property of local mutual exclusion is preserved. Hence, the program will recover to states from where no two neighboring processes are in their critical sections. Once every process enters CS sufficiently many times (equal to the maximum counter value in the initial state), the counter values will be restored, i.e., $\forall j, k : counter.k.j \leq counter.j.j$ will be true. Once counter values are restored, based on Theorem 6.1, for the subsequent computation of the transformed program in read/write model there is an equivalent computation of the given program in WAC model. Thus, we have

Theorem 6.2 If the given program in WAC model is stabilizing fault-tolerant then the transformed program in read/write model is also stabilizing fault-tolerant. □

7 Discussion

Our transformations from (respectively, to) read/write model to (respectively, from) WAC model raises several questions. We discuss some of these questions and their answers, next.

In this paper, we considered transformations of programs in read/write model to WAC model, and vice versa. Are similar transformations possible for other models considered in the literature?

Yes. It is possible to transform a program in shared memory model (where, in an atomic step, a process can read its local state as well as the state of its neighbors and write its own state) or message passing model (where, in an atomic step, a process can either send a message, receive a message, or perform an internal computation) to WAC model, and vice versa. Given a program in shared memory model, first, we can transform it to obtain a corresponding program in read/write model (e.g., [18, 20]). Then, we can use the algorithms in Sections 3 or 4 to transform it into a program that is correct under the WAC model. Moreover, given a program in WAC model, we can use the algorithm in Section 6 to transform it into an algorithm that is correct in read/write model. By definition, it is correct under shared memory model.

Regarding transformation from WAC model to message passing model, we can first use our algorithm from Section 6. Then, we can use the approach in [18] to transform it to a program in message passing model. Likewise, given a program in message passing model, we can first obtain a program in read/write model and then transform it into a program in WAC model.

How efficient are these transformations? And, how does the transformations proposed in this paper compare to others?

In the transformation from read/write model to WAC model, the lower bound on the time required for an enabled process to execute again is $O(d)$, where d is the maximum degree of the communication graph. This is due to the fact that once a process executes, it needs to allow its neighbors to execute before executing again.

Now, we compute the efficiency of the transformations proposed in this paper. For untimed systems, according to Theorem 3.2, at most one process can execute at a time, if processes are deterministic, cannot detect collisions, and cannot perform redundant writes. Hence, the one enabled process may take up to $O(N)$ time before it can execute next, where N is the number of processes in the system. Thus, in untimed systems, the transformation from read/write model to WAC model can slow down the algorithm in WAC model. However, for this model, the delay is inevitable (cf. Theorem 3.2). For timed systems, a process executes once in K slots where K is the number of colors used to color the extended communication graph. Thus, in timed systems, the transformation can slow down the given algorithm by a factor of K that is bounded by $d^2 + 1$, where d is the maximum degree of any node in the graph. This slow down is reasonable in sensor networks where topology is typically geometric and value of K is small (e.g., $K = 5$ for grid-based topology). Table 1 shows the performance of our transformation algorithms.

In [15], the authors introduce *local broadcast with collision* model for sensor networks. They propose *cached sensor transform* (CST) that allows one to correctly simulate a program written for interleaving semantics in sensor networks. Unlike the transformation algorithms proposed in Sections 3 and 4, CST uses CSMA to broadcast the state of a sensor. Furthermore, CST uses randomization while allowing concurrent execution of multiple processes.

In the transformation from WAC model to read/write model (cf. Section 6), the local mutual

Table 1: Performance of different transformation algorithms

Systems assumptions	Time complexity
Untimed (asynchronous) systems	$O(N)$
Timed systems, grid topology	$O(1)$
Timed systems, arbitrary topology	$O(K)$, $K \leq d^2 + 1$

where N is the number of processes in the system, K is the number of colors required to obtain distance-2 vertex coloring, and d is the maximum degree.

exclusion algorithm prevents two neighboring processes from executing concurrently. The slow down caused by this is $O(d)$ where d is the maximum degree in the given communication graph.

Based on the above discussion, these transformations are efficient. However, the issue of optimality in these transformations is open. Since this paper mainly focuses on the *feasibility* of designing such transformations (while preserving the dependability properties of interest) to facilitate the reuse of existing algorithms, the issue of optimality is outside the scope of this paper.

For timed systems, vertex coloring is used to decide when a process can execute. Is edge coloring possible to assign computation slots to each process?

Yes. It is possible to use edge coloring to assign computation slots for each process. Let $f : E \rightarrow [0 \ldots K-1]$ be the color assignments such that $\forall(x, y) \in E : f(x, y) \notin (\{f(j, x) \mid j$ is a neighbor of $x\} \cup \{f(l, y), f(y, l) \mid (x \neq l) \wedge (l$ is a neighbor of $y)\})$. Now, a process (say, x) can execute at slots where $\exists y : y$ is a neighbor of $x : clock.x = f(x, y)$. Moreover, process x can execute at time slots $f(x, y) + c * K$, where K is number of colors used in the network. When x executes, it writes its own state and the state of y. Based on the color assignments, this action does not cause collision at y (although it may cause collision elsewhere).

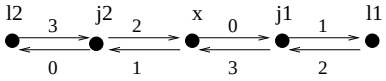

Figure 10: Sample time slots assigned for edges in a network. The number associated with each edge indicates the slot at which a process can write to the other process.

Figure 10 shows a sample computation slot assignment using edge coloring. In this example, process x can execute in slots $0, 1$ and write the state of its neighbors $j1, j2$ respectively. When process x executes in slot 0, process $j2$ can execute and write the state of $l2$. We note that in this case, there will be a collision at processes $j2$ and x. However, this does not interfere with the write actions at processes $l2$ and $j1$. This is due to the fact that collisions do not occur at the receivers. Thus, edge coloring can be used to design transformations for the WAC model.

One of the main drawbacks with edge coloring is energy-efficiency. More specifically, with edge coloring, a process has to execute up to d times in order to update the state of all its neighbors, where d is the number of its neighbors. By contrast, with vertex coloring, a process has to execute only once in order to update the state of all its neighbors. Thus, slot assignment using edge coloring is not energy-efficient when compared to vertex coloring.

In the transformations presented for timed systems, collisions do not occur. Is it possible to obtain transformations where such collisions are permitted?

Yes. We can design transformations even if collisions occur in the system. Specifically, in the above transformation using edge coloring, collisions do occur in the system. However, these collisions do not affect the transformation. In other words, the collisions do not interfere with the write actions of the process as specified by the computation slots.

Are the transformations discussed in this paper possible if the processes can detect collisions?

Yes. The transformations proposed in Sections 3 and 4 are correct even if the processes can detect collisions. However, the optimality of the transformation for untimed systems may not hold. Specifically, in Theorem 3.2, we assume that a process can go from being disabled to being enabled only if its neighbor writes its state. In an algorithm where collision detection is possible, a process can also go from being disabled to being enabled when it detects a collision.

How does the WAC model differ from 'point-to-point message passing' with collision model?

The model considered in this paper is different from 'point-to-point message passing' with collision model. Notably, if we consider a network with four processes, $1, 2, 3, 4$, arranged in a line, then simultaneously, process 1 can send a message to 2 while 3 sends a message to 4. By contrast, in WAC model, if 1 and 3 transmit simultaneously then there will be a collision at 2. We have chosen the WAC model as in many sensor networks (e.g., [4,5]), the only available communication primitive is broadcast to neighbors.

In the transformation shown in Section 6, the value of counter is unbounded. Can it be bounded?

Yes. Although it is preferred to bound the size of variables in stabilizing programs, the definition of self-stabilization does not preclude unbounded counters. Moreover, a simple modification from [34] allows us to bound the counter while preserving stabilization. To bound the counter using the approach in [34], we maintain the bound B on counter to be a number greater than $N^2 + 1$ where N is the number of processes in the system. Now, the counter is incremented modulo B. While the details of this algorithm and the proof of boundedness is beyond the scope of the paper, we describe the modifications needed to bound the counter value.

With this modification, j increments $counter.j.j$ if the counter values of its neighbors are in the (circular) range $[counter.j.j, counter.j.j + N]$. If j and k are neighbors such that the circular difference between $counter.j.j$ and $counter.j.k$ is larger than N and $counter.j.j > counter.j.k$ then j resets its counter to 0. We refer the reader to [34] for the correctness with these bounded counters.

8 Conclusion and Future Work

In this paper, we considered a novel model of computation, *write all with collision* (WAC), and compared it with existing models of computation. The WAC model captures the computation in sensor networks where in an atomic step, a sensor can write its state and the state of its neighbors. However, if two sensors try to write the state of a sensor then none of the writes occur due to collision. Although, our transformation algorithms are designed for the case where collisions are undetectable, as discussed in Section 7, they can be easily used in contexts where collisions are detectable.

We compared the WAC model of computation with other models considered in the literature. We showed that it is possible to transform a program in read/write model into a program in WAC model, and vice versa. However, while transforming a program in read/write model into a program in WAC model, if the transformed program is deterministic and cannot use time then the

transformed program is considerably slow; at most one process can execute at a time. We showed that this is optimal for an untimed, deterministic system where processes cannot detect collisions and cannot perform redundant writes (cf. Theorem 3.2).

To improve the performance of the transformation, we can use one of the following four options. First, we can remove the assumption about the lack of time. Specifically, if the system is timed, we can use the transformations proposed in Section 4. This allows multiple processes to execute concurrently. Second, if the assumption about deterministic processes and time are removed, we can use the TDMA algorithm from [26], where randomization is used. Specifically, in this approach, TDMA slots are assigned using a randomized protocol. Thus, we find that if the processes do not have the ability to read then the ability to determine time consistently is important. Third, if the assumption about redundant writes are removed, more than one process can be allowed to execute simultaneously. Specifically, in Appendix B, we discuss an algorithm for a ring topology using redundant writes. And, fourth, if collisions are detectable, it is expected that efficient transformations can be designed. However, we are not aware of such transformations.

For timed model, if the topology is fixed, our transformation preserves the stabilization property of the given program in read/write model. Further, the transformation of programs in WAC model to read/write model is also stabilization preserving. This transformation does not assume a fixed topology.

We illustrated our transformation algorithm from read/write model into WAC model using the logical grid routing protocol (LGRP) [27, 28] designed for the Line in the Sand project [3]. We transformed the LGRP program in read/write model into a program in WAC model. We showed that the transformed program is similar to the LGRP implementation in [3], where the write-all action is implemented using a broadcast message. Furthermore, the transformed program is the TDMA version of the current LGRP implementation and, hence, the problem of message collision is avoided.

There are several open problems raised by our work. One of these problems is to develop fault-tolerance preserving transformations for WAC model. Since our algorithms require some offline setup (e.g., graph coloring), they cannot deal with topology changes. Based on the results in Section 3, we expect that such transformations would not be possible in untimed, deterministic systems. However, we expect that such transformations could be obtained for timed systems. One of the interesting extension to this work is to design primitives that would allow us to identify transformations where concurrent executions are possible. In this paper, we identified three such primitives, randomization, time and redundant writes. Another interesting extension of this work is to develop programs in WAC model that permit concurrent execution for untimed systems using the knowledge about the speed of processes. Finally, as discussed in Section 7, even though the transformations in this paper are efficient, the issue of optimality remains open.

Acknowledgments. We thank Ted Herman for introducing the WAC model to us at the Seminar on Self-Stabilization, Luminy, France in October 2002.

References

[1] S. S. Kulkarni and M. Arumugam. Transformations for write-all-with-collision model. *In Proceedings of the International Conference on Principles of Distributed Systems (OPODIS), Springer-Verlag*, LNCS:3144:184–197, December 2003.

[2] A. Mairwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. *In Proceedings of the ACM International Workshop On Wireless Sensor Networks and Applications (WSNA)*, 2002.

[3] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y-R. Choi, T. Herman, S. S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks (Elsevier)*, 46(5):605–634, December 2004.

[4] J. Hill, R. Szewczyk, A.Woo, S. Hollar, D. E. Culler, and K. Pister. System architecture directions for network sensors. *In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 2000.

[5] D. Culler, J. Hill, P. Buonadonna, R. Szewczyk, and A. Woo. A network-centric approach to embedded software for tiny devices. In *Emsoft*, volume 2211 of *Lecture Notes in Computer Science*, pages 97–113, 2001.

[6] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

[7] S. Dolev. *Self-Stabilization.* The MIT Press, 2000.

[8] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, April 1997.

[9] M. G. Gouda and M. J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

[10] S. Dolev and J. Welch. Self-stabilizing clock synchronization in presence of Byzantine faults. *Journal of the ACM*, 51(5):780–799, 2004.

[11] A. Daliot, D. Dolev, and H. Parnas. Linear time Byzantine self-stabilizing clock synchronization. *In Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS), Springer*, LNCS:3144:7–19, December 2003.

[12] Y. Chiao, M. Mizuno, and M. L. Neilsen. A self-stabilizing quorum-based protocol for maxima computing. *Distributed Computing*, 15(1):49–55, January 2002.

[13] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self stabilizing protocols for general networks. *In Proceedings of the 4th International Workshop on Distributed Algorithms, Springer*, LNCS:486:15–28, September 1990.

[14] Z. Collin and S. Dolev. Self-stabilizing depth first search. *Information Processing Letters*, 49:297–301, 1994.

[15] T. Herman. Models of self-stabilization and sensor networks. *In Proceedings of the 5th International Workshop on Distributed Computing (IWDC), Springer*, LNCS:2918:205–214, December 2003.

[16] M. Gouda and F. Haddix. The linear alternator. *In Proceedings of the Third Workshop on Self-stabilizing Systems*, pages 31–47, 1997.

[17] M. Gouda and F. Haddix. The alternator. *In Proceedings of the Fourth Workshop on Self-stabilizing Systems*, pages 48–53, 1999.

[18] M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing*, 62(5):766–791, 2002.

[19] H. Kakugawa and M. Yamashita. Self-stabilizing local mutual exclusion on networks in which process identifiers are not distinct. *In Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS)*, pages 202–211, 2002.

[20] G. Antonoiu and P. K. Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph tree that stabilizes using read/write atomicity. *In Euro-par'99 Parallel Processing, Springer-Verlag*, LNCS:1685:824–830, 1999.

[21] K. Ioannidou. Transformations of self-stabilizing algorithms. *In Proceedings of the 16th International Conference on Distributed Computing (DISC), Springer-Verlag*, LNCS:2508:103–117, October 2002.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, September 2001.

[23] G. Chartrand and O. R. Oellermann. *Applied and Algorithmic Graph Theory*. McGraw-Hill Inc., 1993.

[24] S. Ghosh and M. H. Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, 7(1):55–59, 1993.

[25] S. S. Kulkarni and M. Arumugam. SS-TDMA: A self-stabilizing MAC for sensor networks. In *Sensor Network Operations*. IEEE Press, 2005, to appear.

[26] T. Herman and S. Tixeuil. A distributed TDMA slot assignment algorithm for wireless sensor networks. *In Proceedings of the Workshop on Algorithmic Aspects of Wireless Sensor Networks*, Springer, LNCS:3121:45–58, 2004.

[27] L. Miller. Security of the grid routing protocol. Technical Report TR-03-53, The University of Texas at Austin, Department of Computer Sciences, December 2003.

[28] M. G. Gouda and Y-R. Choi. Logical grid routing protocol, 2003. `http://www.cse.ohio-state.edu/siefast/nest/nest_webpage/posters/`.

[29] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[30] N. Lynch and F. Vaandrager. Forward and backward simulations – Part 1: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995. Also, Technical Memo MIT/LCS/TM-486.b, Laboratory for Computer Science, Massachusetts Institute of Technology.

[31] S. Tixeuil. On a space optimal distributed traversal algorithm. *In Proceedings of the Fifth International Workshop on Self-Stabilizing Systems (WSS), Springer-Verlag*, LNCS:2194:216–228, 2001.

[32] J. A. Cobb and M. G. Gouda. Balanced routing. *In Proceedings of the International Conference on Network Protocols (ICNP)*, pages 277–284, October 1997.

[33] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984.

[34] J. Couvreur, N. Francez, and M. Gouda. Asynchronous unison. *In Proceedings of the International Conference on Distributed Computing Systems*, pages 486–493, 1992.

Appendix

A Optimality Issues in Untimed Systems

In this section, we present the proof of Theorem 3.2.

Theorem 3.2 In an untimed system where processes cannot detect collisions, any algorithm that (1) transforms a given program in read/write model into an equivalent program in WAC model under power-set semantics, and (2) ensures that the processes in the generated program in WAC model are deterministic and do not perform redundant writes, must produce programs in WAC model in which at most one process executes at a time.

Proof. As discussed in Section 2, in this proof, we assume that the transformation algorithm does not have any information about the fact that some processes execute only finite number of times in the given read/write program. In other words, the transformation should succeed even if each process executes infinitely often in the given program.

Further, we observe that in WAC model, if the system is untimed and processes are deterministic then a process can go from being disabled to being enabled only if one of its neighboring processes writes its state. Now, consider a program in read/write model that is designed for the system in Figure 11. In the transformed program, assume that j executes at some step. This could allow processes in the left part and the processes in the right part to execute concurrently. Now, we show that if such concurrent execution is permitted then it is impossible to ensure that j can execute once more.

Since c_1 and c_2 may need to write the state of j, the transformation algorithm must ensure that j is disabled at some point after it executes. This will allow one of these neighbors to write the state of j. However, for the write to j to succeed, either c_1 should execute or c_2 should execute, but not both. As we can see, in an untimed, deterministic system, there is no way to ensure that exactly one process from c_1 and c_2 executes if they write the state of j once. Thus, for the system in Figure 11, only one process can execute at a time.

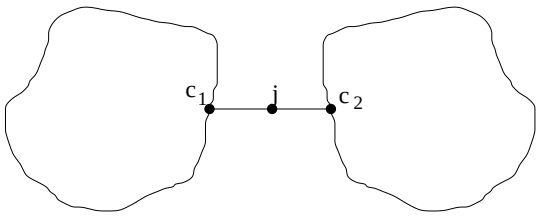

Figure 11: Impossibility of executing more than one process in untimed systems

Moreover, even if there are additional edges (other than $c_1 \leftrightarrow j, c_2 \leftrightarrow j$) among processes in the left network (left of process j), processes in the right network (right of process j), and process j, the above argument still holds. Thus, the theorem holds for all arbitrary connected graphs with at least 3 processes. Note that the proof is trivial for graphs with only 1 or 2 process(es). Thus, an algorithm for transforming programs in read/write model into programs in WAC model, where the system is untimed and processes are deterministic, cannot detect collisions, and cannot perform redundant writes, can allow at most one process to execute at a time. □

B Redundant Writes in Untimed Systems

In this section, we discuss the optimality issues in the transformation, where a program in read/write model is transformed into a program in WAC model, under the assumptions that the system is untimed and processes are deterministic, and cannot detect collisions.

Concurrent executions. We show that it is possible to design transformation algorithms that allow concurrent executions of processes with redundant writes. Specifically, in this section, we present an algorithm where a concurrency of 2 is potentially possible. In this example, initially, processes are mapped onto a logical ring. The communication graph of the original program can have additional links.

For simplicity of presentation, we consider a ring with 6 processes as shown in Figure 12. Let s and t be two special processes. Process s is initially enabled. Whenever s executes, it passes a *token* to the process chains a_1, a_2 (top processes) and b_1, b_2 (bottom processes), allowing them to execute concurrently. In this execution, we guarantee that the write actions of the top processes always succeed. Whenever process t is enabled due to the write actions of the top processes, process t reverses the token circulation direction. In other words, when t executes, it passes the token to the process chains b_2, b_1 and a_2, a_1, thereby allowing them to execute concurrently. In the reverse pass, we guarantee that the write actions of the bottom processes always succeed.

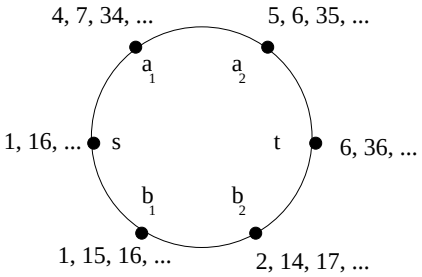

Figure 12: Executing more than one process in untimed systems. The numbers associated with each process denotes the number of writes the process executes in the corresponding round.

Now, we describe how we ensure that the write action of top (respectively, bottom) processes succeed in the forward (respectively, reverse) circulation of token. In the forward pass, when s executes, it writes the state of a_1 and b_1, thereby enabling them. Now, processes a_1 and b_1 can execute concurrently. Since their write actions may collide, we need to ensure that at least one of their writes succeed. To ensure that the writes of the processes in top chain (i.e, a_1, a_2) succeed, when a_1 executes, it writes more than the sum of number of write actions of the processes in the bottom chain (i.e., b_1, b_2). In our solution, when b_1 executes, it writes once, and when b_2 executes, it writes twice. Hence, when a_1 executes, it writes four times, and when a_2 executes, it writes five times. Thus, one or more of the write actions of a_1 and a_2 succeed, thereby enabling successive processes a_2 and t respectively. When the write actions of both a_1 and b_1 succeed at s in the forward pass, s receives confirmation for its write, and hence s cancels its pending writes. When a_2 (respectively, t) writes its state to a_1 (respectively, a_2) successfully, a_1 (respectively, a_2) cancels its pending writes. Now, process t will be enabled since one of the write actions of a_2 would succeed. Note that process t is enabled only by the write action of process a_2, even though the writes of

process b_2 may succeed at t. When process t executes, it reverses the token circulation direction. Further, it initiates the next round, and hence, the number of write actions by each process change accordingly (cf. Figure 12). Moreover, if the write action of t succeeds at b_2 before b_2 executes all its forward writes, b_2 cancels its pending forward writes. And, process t cancels its pending writes when it receives confirmation about its successful writes at b_2 and a_2.

In the reverse pass, we ensure that the writes of the processes in bottom chain succeed. Thus, similar to the above discussion, when a_2 executes, it writes six times, and when a_1 executes, it writes seven times. Hence, when b_2 executes, it writes 14 times, and when b_1 executes, it writes 15 times. Further, process s is enabled only by the write action of b_1, even though the write actions of a_1 may succeed at s. Continuing thus, we can allow two processes to execute concurrently.

We observe that it is not possible to bound the number of write actions by each process in this solution. Further, in this solution, it is possible to have a scenario where only one process is executing at a time. For example, consider the following scenario. Initially, process s executes, and allows process chains a_1, a_2 and b_1, b_2 to execute. However, the bottom chain is slow, and hence, process t becomes enabled by the execution of the top processes. If t executes, writes of the bottom processes may be canceled. Therefore, it is possible to have a scenario where only one process executes at a time. However, as we have discussed above, there is a potential for concurrency.