Transformations for Write-All-With-Collision Model*

Sandeep S. Kulkarni and Mahesh (Umamaheswaran) 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. In this paper, we consider a new atomicity model, *write all with collision* (WAC), and compare it with existing models considered in the literature. This model captures the computations in sensor networks. We show that it is possible to transform a program from WAC model into a program in read/write model, and vice versa. Further, 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. In other words, if a deterministic program cannot read then it is important that it can tell time.

Keywords: Model conversions, Preserving stabilization, Atomicity refinement, Write-all-with-collision model, Read/Write model

1 Introduction

The ability of modeling abstract distributed programs and transforming them into concrete programs that preserve the properties of interest is one of the important problems in distributed systems. Such transformation allows one to write a program in one (typically a simpler/restrictive) model and run it on another (typically a general/less restrictive) model. Hence, several algorithms (e.g., [1–5]) have been proposed for enabling such transformation.

In this paper, we are interested in a new model of computation that occurs in sensor networks. These sensors are resource constrained and can typically communicate with other (neighboring) sensors over a radio network. One of the important issues in these networks is message collision: Due to the shared medium, if a sensor simultaneously receives two messages then they collide and, hence, both 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. Intuitively, in this

* 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

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. (For precise definition, we refer the reader to Section 2.)

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. We would like to note that most of our results are also applicable for the case where collisions can be detected. We discuss this issue in Section 6.

While previous literature has focused on transformations among other models of computation (e.g., [1–5]), the issue of transformation from (respectively, to) WAC model to (respectively, from) other models has not been considered. To redress this deficiency, we focus on the problem of identifying the transformations that will allow us to transform programs in WAC model into read/write model (where a process can either read the state of one of its neighbors or write its own state, but not both), and vice versa.

Contributions of the paper. In this paper, we focus on transformation from WAC model under power-set semantics (where any subset of enabled actions can be executed concurrently) into read/write model, and vice versa. We show that concepts such as graph coloring (e.g., [6–8]), local mutual exclusion (e.g., [4]) and collision-free diffusion [9] can be effectively used for obtaining these transformations. The main contributions of the paper are as follows:

- For untimed (asynchronous) systems, we present an algorithm for the transformation of programs in read/write model into programs in WAC model. We also show the optimality of this transformation; specifically, we show that if the transformed program is deterministic, cannot use time and cannot perform redundant writes (see Section 3 for definition) then at most one process can execute at a time. Also, we argue that this transformation cannot be made stabilization preserving.
- For timed systems, we present an algorithm for the transformation of programs in read/write model into programs in WAC model. This transformation permits concurrent execution of multiple processes. We also show that if the given program in read/write model is stabilizing fault-tolerant [10, 11], i.e., starting from an arbitrary state, it recovers to states from where its specification is satisfied, then, for a fixed topology, the transformed program in WAC model is also stabilizing fault-tolerant. In other words, for timed systems, we show that the transformation is stabilization preserving.
- We present an algorithm for the transformation of programs in WAC model into programs in read/write model. We show that this transformation is also stabilization preserving. This transformation does not assume that the topology is fixed or known in advance.
- We show how to transform programs in other models considered in the literature to WAC model, and vice versa.

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. Then, in Section 3, we present an approach for the transformation of programs in read/write model into programs in WAC model under power-set semantics for untimed systems. Subsequently, in Section 4, we present the transformation for timed systems. Then, in Section 5, we present an approach for the transformation of programs in WAC model under power-set semantics into programs in read/write model. Finally, we make concluding remarks in Section 7.

2 Atomicity Models, Semantics and System Assumptions

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:

$guard \longrightarrow statement$,

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:

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

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:

$predicate(v.j, copy.j, other\_private\_variables.j)$
$\longrightarrow$ 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:

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

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

Semantics. For the WAC model considered in this paper, there are different types of semantics that can be 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 (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. In this paper, unless stated otherwise, we assume that the program in WAC model is executed under power-set semantics.

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. 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. 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).

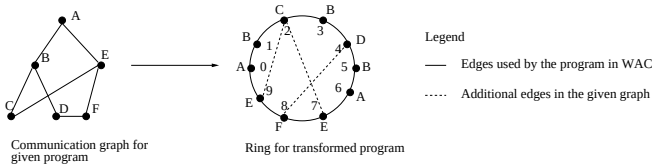

Fig. 1. In the transformed program, 'A' will execute actions of processes 0 and 6, 'B' will execute actions of processes 1, 3 and 5, and so on.

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.

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

Fig. 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.

For reasons of space, we refer the reader to [12] for the proofs of all the theorems in this paper. □

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, in WAC model 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. □

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. In [12], we present a solution that provides potential concurrency if processes are allowed to perform redundant writes. 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.

Stabilization issues. Now, under the assumptions stated in the above theorem, we show that the transformation shown in Figure 2 cannot be stabilization preserving. Based on the assumption that processes do not perform redundant writes, for each process, say j, there exists a local state of that process where none of its actions are enabled. Also, since the guard of an action at process j depends only on local variables of j, 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 algorithm (e.g., [11]) could be used to achieve stabilization in the transformation shown in Figure 2. To add stabilization to the program in Figure 2, 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 algorithms in Section 4.

4 Read/Write Model to WAC Model in Timed Systems

In this section, we present an algorithm for the transformation of a program in read/write model into a program in WAC model for timed systems. Specifically, we present a transformation for an arbitrary topology using graph coloring. Also, this transformation can be achieved using a collision-free time-slot based protocol like time-division multiple access (TDMA) [9, 13]. 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.

Transformation algorithm. 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

3). In other words, two distinct vertices x and y are connected in G' if 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'.

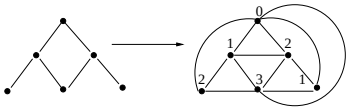

Fig. 3. Transformation using graph coloring. The number associated with each process denotes the color of the process.

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 $\forall c : c \geq 0 : f.j + c * K$. 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 4 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

Fig. 4. Transformation for arbitrary 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. □

Theorem 4.2 If the maximum degree of G is d, then the period between successive executions of a process is at most $d^2 + 1$. □

Remark. The above theorem presents the upper bound on the performance of the transformation. For sensor networks (e.g., [14, 15]), 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 $period = K = 5$.

Preserving stabilization during transformation. 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 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. 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 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. Once the clocks are synchronized, from Theorem 4.1, 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 4.3 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 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., [4]), dining philosophers (e.g., [3,16]) and drinking philosophers (e.g., [3, 16]). 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 [3, 4, 16] 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 5, 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 5 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.

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

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

end

Fig. 5. WAC model to read/write model

Remark. For reasons of space, we refer the reader to [17], for an approach to bound $counter.j$ in the transformation shown in Figure 5.

Theorem 5.1 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. □

Preserving stabilization during transformation. Now, we show that if we use a local mutual exclusion algorithm that is stabilizing (e.g., [3, 4]) then the transformation shown in Figure 5 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 5.1, 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 5.2 If the given program in WAC model is stabilizing fault-tolerant then the transformed program in read/write model is also stabilizing fault-tolerant. □

6 Discussion

Our transformations from (respectively, to) WAC model to (respectively, from) read/write 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., [3, 4]). 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 5 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 5. Then, we can use the approach in [3] 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?

First, we compute the efficiency of the transformation from read/write model to WAC model. 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 this 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, 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).

Likewise, in the transformation from WAC model to read/write model, the local mutual 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.

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.

7 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 6, 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). Also, we showed that, in a timed model, it is possible to allow processes to execute concurrently (cf. Section 4). Thus, we find that if the processes do not have the ability to read then the ability to determine time consistently is important.

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.

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 two such primitives, 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. Another extension of this work is to design transformations for timed systems that allow clock drifts among processes.

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

2. M. Gouda and F. Haddix. The alternator. *In Proceedings of the Fourth Workshop on Self-stabilizing Systems*, pages 48–53, 1999.
3. M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing*, 62(5):766–791, 2002.
4. 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.
5. 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.
6. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, September 2001.
7. G. Chartrand and O. R. Oellermann. *Applied and Algorithmic Graph Theory*. McGraw-Hill Inc., 1993.
8. S. Ghosh and M. H. Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, 7(1):55–59, 1993.
9. S. S. Kulkarni and U. Arumugam. Collision-free communication in sensor networks. *In Proceedings of the Sixth Symposium on Self-Stabilizing Systems (SSS), Springer-Verlag*, LNCS:2704:17–31, June 2003.
10. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.
11. S. Dolev. *Self-Stabilization*. The MIT Press, 2000.
12. S. S. Kulkarni and M. Arumugam. Transformations for write-all-with-collision model. Technical Report MSU-CSE-03-27, Department of Computer Science, Michigan State University, October 2003.
13. W. B. Heinzelman, A. P. Chandrakasan, and H. Balakrishnan. An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions on Wireless Communications*, 1(4):660–670, October 2002.
14. 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.
15. 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.
16. K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984.
17. J. Couvreur, N. Francez, and M. Gouda. Asynchronous unison. *In Proceedings of the International Conference on Distributed Computing Systems*, pages 486–493, 1992.