Noname manuscript No.
(will be inserted by the editor)

Preserving Stabilization while *Practically* Bounding State Space

Vidhya Tekken Valapil · Sandeep S. Kulkarni

Received: date / Accepted: date

Abstract Stabilization is a key dependability property for dealing with unanticipated transient faults, as it guarantees that even in the presence of such faults, the system will recover to states where it satisfies its specification. One of the desirable attributes of stabilization is the use of bounded space for each variable.

In this paper, we present an algorithm that transforms a stabilizing program that uses variables with unbounded domain into a stabilizing program that uses bounded variables and (practically bounded) physical time. While non-stabilizing programs (that do not handle transient faults) can deal with unbounded variables by assigning *large enough but bounded* space, stabilizing programs –that need to deal with arbitrary transient faults– cannot do the same since a transient fault may corrupt the variable to its maximum value.

We show that our transformation algorithm is applicable to several problems including logical clocks, vector clocks, mutual exclusion, diffusing computations, and so on. Moreover, our approach can also be used to bound counters used in an earlier work by Katz and Perry for adding stabilization to a non-stabilizing program. By combining our algorithm with that work by Katz and Perry, it would be possible to provide stabilization for a rich class of problems, by assigning *large enough but bounded* space for variables.

This work is partially supported by NSF CNS 1329807, NSF CNS 1318678, and XPS 1533802.

Vidhya Tekken Valapil
Computer Science and Engineering, Michigan State University, East Lansing,MI-48824,USA.
E-mail: tekkenva@cse.msu.edu

Sandeep S. Kulkarni
Computer Science and Engineering, Michigan State University, East Lansing,MI-48824,USA.
E-mail: sandeep@cse.msu.edu

Keywords Self-stabilization · unbounded variables · bounded space · physical time.

1 Introduction

Self stabilization is one of the highly desirable dependability properties of distributed systems, as it ensures that a system affected by a fault eventually stabilizes to or reaches a valid state in finite time. Stabilizing fault-tolerance is useful for dealing with unexpected transient faults that can perturb the system to a potentially arbitrary state, and guaranteeing recovery to a legitimate state ensures that the effect of these faults would only be temporary.

A key desirable property of stabilizing systems is for them to utilize only variables with a bounded domain. For non-stabilizing systems (i.e. systems that do not handle transient faults), one could utilize counters that grow unbounded by ensuring that the value of the variable remains manageable during the length of the system computation. For example, one could argue that if a variable increases by at most 10 every second and the system can run for at most 1000 seconds then the value would never be more than $10,000$. However, for stabilizing programs, this argument does not hold true. This is because a transient fault could perturb the system to a state where the value has already reached $10,000$ i.e. the bound. The same argument was made by Lamport and Lynch [17] and we recall the argument in the following quote,

> "Simply bounding the number of instance identifiers is of little practical significance, since practical bounds on an unbounded number of identifiers are easy to find. For example, with 64-bit identifiers, a system that chooses ten per second

and was started at the beginning of the universe would not run out of identifiers for several billion more years. However, through a transient error, a node might choose too large an identifier, causing the system to run out of identifiers billions of years too soon–perhaps within a few seconds. A self-stabilizing algorithm using a finite number of identifiers would be quite useful, but we know of no such algorithm."

However, the above quote about unbounded counters conflicts with usage in distributed systems where one often utilizes *time* as a variable whose value is theoretically unbounded. This is because with time there is a guarantee for convergence offered by protocols like NTP, i.e. any inconsistency can be easily detected and corrected in finite time with the help of such protocols. Based on this conflict –we seem to find the use of physical time (that is theoretically unbounded but practically bounded) acceptable but find the use of other unbounded variables unacceptable, so we consider the question: *Why is the usage of unbounded time reasonable, but the usage of other unbounded variables is not?*

To answer this question, we observe that there is an inherent difference between the variable *time* and any other variable. In particular, detecting whether time is corrupted is much easier than detecting if other variables are corrupted. With the usage of redundancy, atomic clocks, etc., one can ensure that the time of a process is *close* to the correct value. In other words, if transient faults perturb a clock to a value that is far away from the current value, this corruption can be detected before using that clock value. Observe that this property of time may not be satisfied by other variables in a given program. For example, if we use logical clocks by Lamport [16], it is possible that the logical clock values of two processes genuinely differ by a large amount. Thus, our goal in this paper is to identify a class of programs for which we can begin with a stabilizing program that relies on unbounded counters and transform it into a program with bounded counters and physical time.

Contributions of the paper.

- We introduce the notion of free and dependent counters and utilize them to develop an algorithm that transforms a stabilizing program with unbounded counters into a stabilizing program with bounded counters.
- We demonstrate that our approach can be combined with that by Katz and Perry [14]. Specifically, [14] provides a mechanism to transform a *non-stabilizing* program into a stabilizing program with unbounded counters. We show that the generated stabilizing program can then be transformed into a stabilizing program that uses bounded counters and physical time.
- We demonstrate our algorithm in the context of classical problems such as diffusing computations, vector clocks and mutual exclusion.
- We show that even with trivially satisfiable parameters for a practical system (like clock drift of less than 100 seconds, messages being delivered or lost within an hour, etc), the size of counters in our transformed programs is small.
- We analyze answers to several questions raised by our work. In particular, we consider certain modifications to our approach for bounding the counters.

Organization of the paper. We define distributed programs and relate their execution with time in Section 2. Our transformation algorithm depends on some assumptions we make about the distributed program at hand, we list these assumptions in Section 2. We define the notion of free and dependent counters in Section 3 and illustrate them with Lamport's logical clocks in Section 4. We present our algorithm and its step-by-step illustration in Section 5 and present its proof of correctness in Section 6. Section 7 demonstrates that our approach can be combined with the approach in [14] by Katz and Perry, so that an existing program can be transformed into a stabilizing program that uses bounded variables and physical time. We discuss additional applications of our algorithm in Section 8. In Section 9, we analyze the effect on the bound of counters derived by our algorithm when clocks (of processes and global clock) differ from each other by more that one region. Section 10 discusses related work and questions raised by our work. We present concluding remarks and future work in Section 11. We present a summarized table of notations used in this paper in the Appendix.

2 Preliminaries

2.1 Modeling Distributed Programs

A distributed program consists of a set of processes, where each process has a set of actions. The program executes in an interleaving manner where an action of some process is executed in every step. Execution of the program is captured with a set of program variables, each of which is associated with a domain. With this intuition, we now formally define a program in terms of its variables and actions. Definitions 1-8 are from standard literature [5, 7, 12].

Definition 1 (Program). A program p is of the form $\langle V_p, A_p \rangle$, where V_p is a set of variables, and A_p is a set of

actions that are of the form *guard* $\rightarrow$ *statement*, where *guard* is a condition involving the variables in V_p and the *statement* updates a subset of variables in V_p.

Definition 2 (State). A state s of program p is obtained by assigning each variable in V_p a value from its domain.

Definition 3 (Value of a Program Variable). Let s be a state of of program p and let $v \in V_p$ be a variable of p. We use $v(s)$ to denote the value of variable v in state s.

Definition 4 (Enabled). An action of the form *guard* $\rightarrow$ *statement* is enabled in state s iff the *guard* evaluates to true in state s.

Although our transformation algorithm works with any daemon/semantics, for the sake of simplicity, in our exposition, we use interleaving semantics and define the step of a program as follows:

Definition 5 (Step/Transition). Let s_0 and s_1 be two states of program p. (s_0, s_1) is a step/transition of p iff s_1 is reached by executing the statement of an enabled action in state s_0.

Definition 6 (Computation). A computation of program p is a sequence of states $s_0, s_1, \cdots$, such that $\forall l, l \geq 0, (s_l, s_{l+1})$ is a transition of p.

Remark 1 For the sake of simplicity, we assume that there is at least one enabled action in state s. If such an action does not exist, we pretend that the program has an action corresponding to a self-loop in state s.

Definition 7 (Computation Prefix). A finite sequence of states $s_0, s_1, \cdots, s_n$ is a computation-prefix of program p iff it is a prefix of some computation of p.

Finally, we recall the definition of stabilization from [7]:

Definition 8 (Stabilization). Program p is stabilizing to S, where S is a set of states, iff
- Starting from an arbitrary state, every computation of p reaches S, and
- Starting from a state in S, no computation of p reaches a state outside S.

2.2 Modeling Practical Systems: Clock Synchronization and Speed of Processes

Our transformation algorithm relies on (theoretically unbounded) time to add stabilization. The definitions in Section 2.1 are time-independent. They also do not involve the notion of processes, time/clock synchronization among them, or number of actions that could execute in a given time. In this section, we introduce these concepts to define $\langle \mathcal{RS}, max_{inc} \rangle$ system (cf. Definition 10) that captures what we mean by a *practical* distributed program.

A practical distributed program, p, consists of a set of processes, pr_p. Typically, the variables V_p of program p are split so that each process can read/write a subset of them. Our work does not rely on the structure of this distribution. Rather, we only assume that each process, say j, is associated with time t_j which is a variable of p. As far as the program is concerned, t_j is read-only. We also introduce an abstract global time t_g that **cannot** be read or written by program p. Let T_p denote the set of clock variables in program p.

Remark 2 Note that we are adding t_g to V_p. This is only intended to help us get a global perspective of the program computation and to reason about abstract global time associated with state s i.e., to utilize notations such as $t_g(s)$ to capture the value of the abstract global time in state s. We assume that, syntactically, actions of program p cannot read or write t_g. Likewise, actions of process j in p cannot read t_k, $k \neq j$. And, actions of p can only read t_j,

With the introduction of time variables to the program, we extend Definition 5 as follows,

Definition 9 (Step/Transition in a timed program). Let s_0 and s_1 be two states of program p. (s_0, s_1) is a timed step/transition of p iff s_1 is obtained by

- executing the statement of an enabled action in state s_0,
- changing variables in $T_p - \{t_g\}$ in any arbitrary way, and
- increasing the value of t_g, i.e., ensuring that $t_g(s_1) > t_g(s_0)$.

Note that the above definition allows clock of zero or more processes to change simultaneously. This is expected because as time elapses it is possible for time associated with multiple processes to change simultaneously in a timed step. We permit clocks of processes to even go backwards like it can happen with NTP clocks. However, these clocks are synchronized partially using protocols like NTP. We discuss this aspect next.

In a practical distributed program, clocks at processes are close to the abstract global clock and to each other. Furthermore, there is a limit on the total increase in the value of any variable in the program within a specific duration of time. To capture this intuition, next,

we define the notion of $\langle \mathcal{RS}, max_{inc} \rangle$-distributed system, where RS is a duration of time and max_{inc} is an integer that provides a bound on the maximum increase in the value of any variable within duration $\mathcal{RS}$.

More specifically, we partition abstract global time into **regions** of size $\mathcal{RS}$, so at global time t_g the corresponding (global) **region** is identified by $\lfloor \frac{t_g}{\mathcal{RS}} \rfloor$. Likewise, for each process j, its time t_j is also mapped to a region i.e., the region of process j is $\lfloor \frac{t_j}{\mathcal{RS}} \rfloor$.

Definition 10 We say that a program p is running in a $\langle \mathcal{RS}, max_{inc} \rangle$-system iff for any computation $\langle s_0, s_1, \cdots \rangle$ of p, the following properties are satisfied.

1. At a given state say s_l, for any process j $|\lfloor \frac{t_g(s_l)}{\mathcal{RS}} \rfloor - \lfloor \frac{t_j(s_l)}{\mathcal{RS}} \rfloor| \leq 1$
2. At a given state say s_l, for any two processes j, k $|\lfloor \frac{t_j(s_l)}{\mathcal{RS}} \rfloor - \lfloor \frac{t_k(s_l)}{\mathcal{RS}} \rfloor| \leq 1$
3. For any two states say s_a and s_b and any variable $v \in V_p - T_p$, $(\lfloor \frac{t_g(s_a)}{\mathcal{RS}} \rfloor == \lfloor \frac{t_g(s_b)}{\mathcal{RS}} \rfloor) \implies (v(s_b) - v(s_a)) < max_{inc}$

In the above definition, $\lfloor \frac{t_g(s_l)}{\mathcal{RS}} \rfloor$ and $\lfloor \frac{t_j(s_l)}{\mathcal{RS}} \rfloor$ denote the abstract global region associated with state s_l and region of process j in state s_l. $\mathcal{RS}$ should be chosen so that (1) the region of any process differs from the abstract global region by at most 1, and (2) regions of any two processes differ from each other by at most 1. Furthermore max_{inc} should be chosen such that within a region i.e., within $\mathcal{RS}$ the maximum increase in the value of any variable in the program is max_{inc}.

We can also extend the notion of $\langle \mathcal{RS}, max_{inc} \rangle$- system to *eventually* $\langle \mathcal{RS}, max_{inc} \rangle$- *system* by requiring that the above properties are satisfied for a suffix of computation of p. In the context of stabilizing programs, we assume that the given program is running in a $\langle \mathcal{RS}, max_{inc} \rangle$- system for some known value of $\mathcal{RS}$ and max_{inc}. However, the above properties may be violated momentarily due to transient faults (e.g., clocks of two processes may have a large difference). We assume that these properties will be restored eventually, i.e., the program will eventually be running in a $\langle \mathcal{RS}, max_{inc} \rangle$-system after the occurrence of faults.

For a program running in a distributed system, its designer can determine and provide the values of $\mathcal{RS}$ and max_{inc} based on the underlying clock synchronization and other design guarantees. For example, for a system with clock synchronization guarantee of say 10s (i.e., clocks of any two processes differ from each other by at most 10s), the region-size $\mathcal{RS}$ could be 10s or more. Similarly, for a program running in a system where an event happens every nanosecond and there are say 10 processes, so there are at most 10 events per nanosecond, within a chosen region-size say $\mathcal{RS}$ =10s there can be at most $(10^9) * 10 * 10$ events, and if the value of any variable in the program is guaranteed to increase by at most 1 at each event then $max_{inc} = 10^{11}$ will satisfy Condition 3 in Definition 10.

Assumption 1. *In the rest of the paper, we assume that the input program (i.e., the given stabilizing program with unbounded variables) is running in a $\langle \mathcal{RS}, max_{inc} \rangle$-system. Furthermore, unless specified otherwise, we let $RS = 100s$, i.e., clocks are synchronized to be within $100s$. This is reasonable for most if not all systems, given that protocols like NTP provide synchronization that is within a few milliseconds. The value of max_{inc} would be selected for each program based on its analysis.*

3 Free Counters and Dependent Counters

In this section, we define the notion of free and dependent counters to characterize variables of p other than those in T_p (clock variables of p). This notion forms the basis of our transformation algorithm. However, before we do that, we focus on the structure of the variables in the program.

In particular, for program p, we partition its variables V_p into two types: simple variables and complex variables. Simple variables are those variables with domain that is either a finite set or N, the set of natural numbers. And, complex variables are *collections* (e.g., set, sequence, list, etc.) of simple variables, and the constituent variables can be removed/added dynamically. To define the notion of free and dependent counters, we will unravel the structure of a complex variable and focus only on the simple variables contained in it. For example, if the program contains a complex variable, say C, which is a set and its current value is $\{3, 5, 7\}$, then we visualize this as having three simple variables c_1, c_2 and c_3 whose values are 3, 5 and 7 respectively.

With this intuition, we can view a program p with variables V_p as an equivalent program with variables SV_p, where SV_p is a dynamically changing collection of simple variables. Moreover, the domain of any variable in SV_p is either finite or equal to N, the set of natural numbers.

Remark 3 A reader might wonder why we do not define program p in terms of SV_p in the first place. SV_p is a dynamic set that has a flexible size. To update a dynamic set of variables one would require a dynamic (infinite) set of actions. Without making explicit efforts, such a model has potential to model programs that are not recursively enumerable. Our modeling with complex variables in V_p avoids this problem, as the set of actions is always finite.

The set SV_p is dynamic. We say that a variable in SV_p is a **permanent variable** if it is guaranteed to be present in every state of p. For example, any simple variable in V_p would be a permanent variable since it will be present in SV_p at all times. A variable that is not a permanent variable is called a **temporary variable**.

We overload Definition 3 for SV_p. Specifically, if variable v is present in SV_p in the given state, the value of that variable is defined in the same manner as in the Definition 3. And, if the variable v is not present in that state (entries in complex variables in V_p or their equivalent simple variables in SV_p may be added/removed), we denote its value as $\perp$. In other words,

Definition 11 (Valuation of variable in SV_p). Let v be a variable in SV_p and let s be a state of program p. If v is present in state s, then $v(s)$ denotes the value of v in state s. And, if v is not present in s, then we denote it as $v(s) = \perp$.

With the help of SV_p and permanent/temporary variables, we define the notion of free and dependent counters. Intuitively, a free counter is a permanent variable whose value never decreases. Moreover, if we increase the value of the free counter in the final state of a computation-prefix then the resulting sequence is also a valid computation prefix of the given program. Formally,

Definition 12 (Free counter). A permanent variable fc of program p running in a $\langle \mathcal{RS}, max_{inc}\rangle$-system is a free counter iff for any computation prefix $\rho = s_0, s_1, s_2, \cdots, s_l$ of p the following conditions hold:

(i) $\forall w : 0 \leq w < l : fc(s_{w+1}) \geq fc(s_w)$,

(ii) $\rho' = \rho + s_{l+1}$ is also a valid computation prefix, where state s_{l+1} is reached from state s_l by increasing the value of fc (and leaving other variables except those in T_p i.e., clock variables in p, unchanged), and $\rho + s_{l+1}$ denotes the concatenation of ρ and s_{l+1}.

Remark 4 Note that the above definition requires that the value of the free counter never decreases. Also, if $\rho = s_0, s_1, \cdots, s_l$ is a valid computation prefix of p, appending state s_{l+1} results in another valid computation prefix if (1) $fc(s_{l+1}) = fc(s_l) + d$, and (2) T_p is updated as permitted by Definition 9. Since we are assuming that the input program, p, is running in a $\langle \mathcal{RS}, max_{inc}\rangle$-system, there is a limit to the increase of a free counter in the computation of p in a given duration of time. As we discuss later in Section 5, the ability to increase free counters is used to perform the transformation and obtain program p' that preserves stabilization of p while using bounded counters.

Next, we define the notion of dependent counters. A dependent counter is a temporary variable. We require that when this variable is created/added, its value is set to the value of some free counter within at most r_b preceding **(global) regions**. Moreover, after r_f **(global) regions**, this temporary variable is removed. And, in between the value remains unchanged.

Thus, we treat a variable or counter as (r_b, r_f)-dependent counter if (1) when the variable is set to a value different from $\perp$ (i.e. when it is created), it is set to the value of some free counter in at most r_b (global) regions in the past, and (2) after the value of the variable is set to a value different from $\perp$, within r_f (global) regions it is set back to $\perp$ (i.e., when it is removed). Hence, we define dependent counters as follows:

Definition 13 (Dependent counter). A temporary variable dc of program p running in a $\langle \mathcal{RS}, max_{inc}\rangle$-system is a (r_b,r_f)-dependent counter iff for any computation $\rho = s_0, s_1, s_2, \cdots$ of program p the following conditions hold: $\forall a : a \geq 0 :$

1. $dc(s_a) = \perp \wedge dc(s_{a+1}) \neq \perp$
$\Rightarrow \exists w : (w \leq a+1) \wedge dc(s_{a+1}) = fc(s_w) \wedge$
$(\lfloor \frac{t_g(s_{a+1})}{\mathcal{RS}} \rfloor - r_b) \leq \lfloor \frac{t_g(s_w)}{\mathcal{RS}} \rfloor \leq \lfloor \frac{t_g(s_{a+1})}{\mathcal{RS}} \rfloor$
2. $dc(s_a) \neq \perp$
$\Rightarrow \forall w : (\lfloor \frac{t_g(s_a)}{\mathcal{RS}} \rfloor + r_f) \leq \lfloor \frac{t_g(s_w)}{\mathcal{RS}} \rfloor : dc(s_w) = \perp$
3. $dc(s_a) \neq \perp \wedge dc(s_{a+1}) \neq \perp \quad \Rightarrow \quad dc(s_a) = dc(s_{a+1})$

Remark 5 Note that this requirement (i.e. when the variable is created it is set to the value of some free counter within at most r_b preceding regions) is not restrictive, because essentially, the requirement is just that the value assigned to the dependent counter is somehow related to a free counter in the recent past. For example, if variable dc is set to $fc-5$ where fc is a free counter, then we can treat it as having two variables $dc1$ and $dc2$, where setting dc to $fc-5$ is modeled as setting $dc1$ to be same as fc (free counter in the same/current region) and $dc2$ to -5, and using $dc1 + dc2$ instead of dc. Note that the latter is a bounded variable whereas the former can be used to satisfy the requirements of dependent counters. Likewise, setting dc to $2 * fc$ or $fc^2 + 10$ would be acceptable as well. Since there are too many such choices, to keep the transformation algorithm simple, we use the above definition. However, in practice some *syntactical* tweaking of a given program without affecting its properties might be required.

Remark 6 Goal of the second requirement (i.e. after r_f regions the variable is removed) is that the value of the counter will eventually become obsolete and hence will no longer affect the program execution. We discuss this further in Section 8, where this requirement is handled/satisfied by syntactical changes to a given program.

Remark 7 For a given program, irrespective of the kind of collection (a set or a list or a sequence) that a complex variable may correspond to, our algorithm focuses only on bounding each constituent simple variable or entry in the complex variable, whereas the overall structure or the complex variable itself remains unaffected by the algorithm. In other words, operations associated with the data structure itself (e.g., next element in the list) are performed as is. However, any operation on the data item (e.g., if first item in the list is equal to 0) would be affected by our transformation algorithm. In this case, based on the properties of that list item we transform it using our algorithm before the equality operation is performed.

4 Illustrating Free and Dependent Counters

In this section, we illustrate our definitions of free and dependent counters with the help of Lamport's logical clocks [16]. In this program, the processes in the program communicate through messages. We assume that any message reaches its destination within v (global) regions. As mentioned in Assumption 1, we consider $RS = 100s$. Furthermore, in most if not all practical systems, we can guarantee that a message is received within an hour ($3600s$). This will allow us to select $v = 36$. At any point in time each process j has a logical clock value $cl.j$ associated with it, and $cl.j$ increases whenever an event occurs at j. Next, using our formalism from Section 2.1, we specify the actions of this program. Also, we identify the notion of simple versus complex variables, dependent versus free counters, etc. The actions of a process, say j, in this program are as follows:

1. Action Local Event
   $true \longrightarrow cl.j = cl.j + d;$
2. Action Send Event, say a message m is sent to process k
   $true \longrightarrow cl.j = cl.j + d; cl.m = cl.j;$
   $channel_{j,k} = channel_{j,k} \cup \{m\}.$
3. Action Receive Event, say a message m was received from process k
   $m \in channel_{j,k} \longrightarrow cl.j = max(cl.j, cl.m) + d;$
   $channel_{j,k} = channel_{j,k} - \{m\}$

where d is any positive integer that can be different at different instances of the actions. Observe that for every process j, $cl.j$ is a permanent variable. The variable $channel_{j,k}$ is a *complex* variable which contains timestamps of messages in transit ($cl.m$ denotes the timestamp of message m in transit). If we unravel this variable, we get multiple timestamps, each corresponding to a message in transit.

In Lamport's logical clock program,

1. $cl.j$ is a free counter
   Proof. The permanent variable $cl.j$ is a free counter of process j that satisfies Definition 12. In particular if $\langle s_0, s_1, s_2, \cdots, s_n\rangle$ is a computation prefix of p, then:
   (i) at a given state s_l when an event occurs, the value of $cl.j$ is computed as $cl.j(s_l) = cl.j(s_{l-1}) + d$, $d > 0$, or $cl.j(s_l) = max(cl.j(s_{l-1}), cl.m) + d$, $d > 0$, i.e., it is higher than the logical clock value of j in its previous state s_{l-1}. Thus $cl.j$ is an unbounded counter that has the form $cl.j(s_l) > cl.j(s_{l-1})$ i.e. it never decreases.
   (ii) Also, if $\rho = \langle s_0, s_1, s_2, \cdots, s_l\rangle$ is a valid computation prefix of p, appending state s_{l+1} to ρ, where $cl.j(s_{l+1}) = cl.j(s_l) + d$ results in $\rho' = \rho + s_{l+1}$ which is also a valid computation prefix. In other words, an increase in the logical clock value of a process by d continues to preserve the correctness of the overall program. Thus the logical clock value associated with any process is a *free counter*.
2. Each entry in $channel_{j,k}$ is a $(0,v)$- dependent counter provided any message is guaranteed to be received within v (global) regions.
   Proof. Entries in $channel_{j,k}$ i.e., message timestamps are *dependent counters* in the program, since they are temporary variables that have the form outlined in Definition 13. Let $\langle s_0, s_1, \cdots\rangle$ be a computation of p and $cl.m$ denote the timestamp of a message m in $channel_{j,k}$. Then, $cl.m(s_j) = \perp$ when message m is not in transit (before transmission or after reception) and $cl.m$ corresponds to the timestamp of message m when m is in transit.
   (i) if there exists a state s_a such that $cl.m(s_a) = \perp$ and $cl.m(s_{a+1}) \neq \perp$ then this corresponds to sending of message m. In this case, $cl.m(s_{a+1})$ is set to $cl.j(s_{a+1})$, which is the value of a free counter in the same state therefore the same (global) region. It follows that this satisfies condition 1 of Definition 13, where $r_b = 0$ in this scenario.
   (ii) if $cl.m(s_a) \neq \perp$ in some state s_a then it means that message m is in transit in state s_a. Let r_e denote the (global) region corresponding to state s_a, i.e., $r_e = \lfloor \frac{t_g(s_a)}{\mathcal{RS}} \rfloor$. Since we assume that every message would be delivered in v regions, it follows that after v (global) regions, in any state s_b in (global) region (r_{e+v}) or greater, message m will no longer be in transit, $cl.m(s_b) = \perp$. This satisfies condition 2 of Definition 13, where $r_f = v$ in this scenario.
   (iii) a message m is timestamped only once, i.e. when it is added to $channel_{j,k}$ and $cl.m$ is set to $\perp$ only after it is removed from $channel_{j,k}$. When m is

in transmission the value of $cl.m$ is never changed. This satisfies condition 3 of Definition 13.

Remark 8 Continuing our discussion about free counters from Remark 4, observe that $cl.j$ is a free counter does not imply the program continuously increases the value of $cl.j$. If a program simply keeps doing that, its computation may violate the requirements of $\langle \mathcal{RS}, max_{inc} \rangle$-system. In particular, it may increase $cl.j$ beyond the expected maximum increase (max_{inc}). However, it does imply that we can increase $cl.j$ (and create a local event) when needed without affecting the correctness of the program and we use this ability (of free counters) in our transformation algorithm to bound program counters.

5 Transformation Algorithm

The input to our algorithm is a program p running in a $\langle \mathcal{RS}, max_{inc} \rangle$-system that is stabilizing under the assumption that the free and dependent counters can grow unbounded. Our algorithm translates p into program p' such that execution of p' in that system would be stabilizing and would use bounded counters and (practically) bounded physical time. Our algorithm utilizes the observation that while counters used in a program can grow unbounded, their maximum growth in a given period of time (assuming no transient faults) can be computed.

We proceed as follows: In Section 5.1 we illustrate the first step of our algorithm in the context of Logical clocks (from Section 4). In Section 5.2 we generalize the discussion in Section 5.1 to obtain Step 1 of our algorithm. We present Step 2 and Step 3 of our algorithm in Sections 5.3 and 5.5, and their corresponding illustrations in Sections 5.4 and 5.6 respectively.

5.1 Illustration of the Step 1 for Logical Clocks

In this section, before we describe our approach, we illustrate it in the context of logical clocks by Lamport from Section 4. Note that in this program, for any process j its logical clock $cl.j$ is a free counter. For the sake of illustration, let us assume that the counter is incremented by one in any action. And, there are at most 10 events in one region. In other words, the program is running in a $\langle \mathcal{RS}, max_{inc} \rangle$-system where $max_{inc} = 10$. Furthermore, assume that initially, all values of $cl.j$ are -1. Initially, region of every process is -1. As soon as it creates the first event, it is in region 0. Thus, in one $\mathcal{RS}$ time, value of $cl.j$ would increase to at most 10. In $2\mathcal{RS}$ time, it would increase to at most 20 and so on.

I. Assumption
The input program p is running in a $\langle \mathcal{RS}, max_{inc} \rangle$-system, where max_{inc} is the maximum increase in any free counter in one global region

II. Variables:
$r.j$: region of process j computed as $\lfloor \frac{t_j}{\mathcal{RS}} \rfloor$
$fc.j$: free counter of process j
$dc.j$: $\langle r_b, r_f \rangle$ dependent counter of process j
max_r : maximum $(r_b + r_f)$ value for dependent counters
$MAXBOUND = 3 * [max_{inc} * (11 + 3 * max_r)]$
$minfree = 3 * r.j * max_{inc}$
$maxfree = 3 * (r.j + 1) * max_{inc} + 2 * max_{inc} - 1$
$mindep = 3 * (r.j - 2 - max_r) * max_{inc}$
$maxdep = 3 * (r.j + 1) * max_{inc} + 2 * max_{inc} - 1$

III. Whenever the region associated with process j changes:
1. For each free counter $fc.j$ in j:
   $intfc = convertfc(fc)$
   checkfc($intfc.j, r.j$)
   set $fc = intfc \ mod \ MAXBOUND$
2. For each dependent counter $dc.j$ in j:
   $intdc = convertdc(dc)$
   checkdc($intdc.j, r.j$)
   set $dc = intdc \ mod \ MAXBOUND$

IV. For each action $guard \rightarrow statement$ **of process** j
   // transformed program maintains free/dependent
   // counters in a modulo form. First convert it to
   // corresponding integer format.
1. For each free counter fc in the *guard*:
   $intfc = convertfc(fc)$
2. For each dependent counter dc in the *guard*:
   $intdc = convertdc(dc)$
3. Evaluate *guards* with updated counters and select an action to execute
   // Note: original program actions utilize unbounded counters.
4. Whenever $intfc$ is updated in the statement
   checkfc($intfc, r.j$)
   set $fc = intfc \ mod \ MAXBOUND$
5. Whenever $intdc$ is updated in the statement
   checkdc($intdc, r.j$)
   set $dc = intdc \ mod \ MAXBOUND$

V. Function checkfc($fc, r.j$)**:**
If ($(fc < minfree)$ || $(fc > maxfree)$) then:
   set $fc := minfree$ *//reset free counter to minimum value in the legitimate range*

VI. Function checkdc($dc, r.j$)**:**
If ($(dc < mindep)$ || $(dc > maxdep)$ then:
   set $dc := mindep$ *//reset dependent counter to minimum value in the legitimate range*

VII. Function convertfc(x)**:**
Find y such that $x = y \ mod \ MAXBOUND$ and y is in range [minfree .. maxfree]
If no such y exists, return minfree

VIII. Function convertdc(x)**:**
Find y such that $x = y \ mod \ MAXBOUND$ and y is in range [mindep .. maxdep]
If no such y exists, return mindep

Fig. 1 Our Transformation Algorithm for process j in the given program

Our first attempt to revise this program would be to require that in region 0 the value of $cl.j$ would be between [0..9]. In region 1, $cl.j$ would be between [10..19] and so on. Observe that the first property is already satisfied by the original program. However, the second property may be violated since $cl.j$ may be less than 10 even in region 1. We can remedy this by increasing the value of $cl.j$ as needed. Note that in this instance,

the fact that $cl.j$ is a free counter is important, as it guarantees that $cl.j$ will never decrease and we are permitted to increase $cl.j$ as needed. With region 1, we need to ensure that $cl.j$ does not increase beyond 19, as we are not allowed to decrease it. We can try to ensure this property by the length of computation which guarantees a bound on the growth in the value of $cl.j$ in region 1.

While the above approach is reasonable, it suffers from a problem that processes do not always agree on what the current region is. For e.g. process j could be in region 1 but process k could still be in region 0. Now, j could send a message to k causing k to have a value of $cl.k$ that is outside [0..9].

Also, if process j moves quickly to region 1 while process k is still in region 0 then it creates some additional difficulties. In such a system, the clock synchronization may force j to *slow down* its clock to ensure that k can catch up. (An alternative is to let process k advance its clock more quickly. But we assume that we do not control the clock synchronization algorithm.)

In other words, as far as process j is concerned, even if it starts with initial value of $cl.j = 10$ at the beginning of region 1, the value of $cl.j$ may exceed 19 before j enters region 2. We can remedy these problems with the observation that region of two processes differs by at most 1. Hence, even if clock of j is forced to slow down to let k catch-up, as long as process j is in the same region, $cl.j$ will not increase by more than 30. (Note that the value 30 = 10*3 is due to the fact that region 1 of process j can overlap with global regions $0, 1$ and 2 and in each global region the increase in $cl.j$ is bounded by 10.)

With this approach, we proceed as follows: In region 0, we *try to* ensure that the value of $cl.j$ is between [0..29], in region 1, the value of $cl.j$ is between [30..59], in region 2, value of $cl.j$ is between [60..89] and so on. Observe that with this change, when the first process, say j, moves to region 1, all values were less than 30. Based on assumption about bounded growth in the region, as long as process j is in region 1, value of cl cannot increase beyond 59.

We can generalize the above approach by the constraint that if the region of process j equals r then *we try to* ensure that $cl.j$ is between $[30r..30r + 29]$. However, while process j is in region r, process k could move to region $r + 1$ thereby creating a value that is greater than $30r+29$. Moreover, if process k communicates with process j, it could force process j to increase $cl.j$ to be more than $30r + 29$. However, as long as process j is in region r and process k is in region $r + 1$ (which can last for at most $2\mathcal{RS}$ time), values of $cl.j$ or $cl.k$ cannot increase beyond $30r + 29 + 20$.

Based on this observation, we define r_{min} and r_{max} that identifies the minimum region value held by some process and maximum region value held by some process. (Note that these values differ by at most 1. Furthermore, processes are not aware of these values. They only know that the value of their region equals one of them.) So, we observe that the value of $cl.j$ is in the interval $[30r_{min}..30r_{min} + 29 + 20]$. Moreover, it is also guaranteed to be in $[30r_{max} - 30..30r_{max} + 19]$. In addition, due to the property of regions, at some point, all processes must be in the same region. (If some processes are in Region $r - 1$ and some are in region r, then no process can move to region $r + 1$ as long as some process is in Region $r - 1$. Thus, just before the first process moves to region $r + 1$, all processes must be in the same region, namely region r.) Let this region be r. Clearly, r_{min} and r_{max} are equal to r at this time. From the above discussion, at this point, $cl.j$ must be in $[30r..30r + 19]$.

The above analysis is correct if we assume that the value of $cl.j$ started with initialized values. However, if the values are corrupted, the above property may not hold. To handle this, we change the value of $cl.j$ when we know that it is corrupted. For example, let process j be in region r. Since r_{min} is at least r, the value of $cl.j$ is at least $30r$. Also, since r_{max} is at most $r+1$, the value of $cl.j$ is at most $30(r + 1) + 19$. If process j finds itself in a situation where the value of $cl.j$ is outside this domain, then it resets it to $30r$.

Additionally, as discussed above, there exists a time when all processes are in the same region. Given the local correction action to ensure that $cl.j$ is in the range $[30r..30(r + 1) + 19]$, it follows that when the first process is about to move to region $r + 1$, the highest cl value of any process is $30(r + 1) + 19$. When the first process moves to Region $r + 2$ (overlap can be with at most two global regions), the increase can be up to 20 so the highest cl value is $30(r + 2) + 9$. Using the same argument, when the first process moves to Region $r+3$, all cl values are less than $30(r+3)$. Thus, when all processes move to Region $r + 3$, their cl values are in the range $[30(r + 3)..30(r + 3) + 29]$. This property is preserved for all future regions. Observe that this implies that within 3 regions, the value of free counters is within these expected ranges. The above discussion rested on the assumption that $max_{inc} = 10$ i.e.,the maximum increase in the value of any counter within a region is at most 10. Next, we generalize this to obtain the first step of our algorithm.

5.2 Algorithm for Step 1: Adjusting Free Counters

As free counters can be increased at will, the natural approach is to try to keep the value of the free counter in region r to be $[r * max_{inc}..(r+1) * max_{inc}]$. It turns out that this is not feasible since the process regions may not be identical. So, a process in region $r+1$ may send a message to process in region r causing it to receive values that are outside this range. Hence, in our algorithm, we proceed as follows: we *try to* ensure that in region r, the value of any free counter is in the range $[3r*max_{inc}..3(r+1)*max_{inc}-1]$. However, in practice, since the regions of two processes may not be identical, we will ensure that the value of the free counter is in the range $[3r * max_{inc}..3(r+1) * max_{inc} + 2 * max_{inc} - 1]$. Each process will first ensure that this constraint is satisfied. If it is not, it will restore the value of the free counter to $3r * max_{inc}$, where r is *its* current region. This can be achieved by checking the values of the free counter (1) as soon as the region of the process changes (III.1 in Figure 1), or (2) the process updates its free counter as part of its actions (IV.4 in Figure 1), or (3) process uses the free counter (in evaluating guard of an action) (IV.1 in Figure 1).

5.3 Algorithm for Step 2: Adjusting Dependent Counters

Let dc be a (r_b, r_f) dependent counter. Now, we identify the possible values of dc that may happen under legitimate states, i.e., in the absence of faults. Consider the case where a process is in region r, the value of its free counter is in the range $[3*r*max_{inc}..3*(r+1)*max_{inc}+2*max_{inc}-1]$. At this time, the global region is at least $r-1$. If the counter dc is used in global region $r-1$, then it was initialized in global region greater than or equal to $r-1-r_f$. Moreover, the value it was set to can only come from a free counter r_b regions earlier. In other words, the value of dc was set to the value of a free counter in global region $r-1-r_b-r_f$ or higher. Since the process region and global region may differ by 1, the region of the process that set the value of dc is at least $r-2-r_b-r_f$. Hence, the value of dc is at least $3*(r-2-r_b-r_f)*max_{inc}$ Moreover, the maximum value of dc is the maximum value of some free counter, i.e., it is $3*(r+1)*max_{inc}+2*max_{inc}-1$.

Hence, in Step 2 of our algorithm, we ensure that the value of a given dependent counter is always within this range. If it is not in this range, we set it to the minimum permitted value in this range, i.e., we set it to $3*(r-2-r_b-r_f)*max_{inc}$. Similar to free counters, this checking is done (1) as soon as the region of the process changes (III.2 in Figure 1), or (2) when the process sets a dependent counter as part of its actions (IV.5 in Figure 1), or (3) the process uses the dependent counter (in evaluating guard of an action) (IV.2 in Figure 1). Each dependent counter is characterized by parameters r_b and r_f. Let the maximum value of $r_b + r_f$ for any dependent counter be max_r, Thus, if a process is in region r, its dependent counters must be in the range $[3*(r-2-max_r)*max_{inc}..3*(r+1)*max_{inc}+2*max_{inc}-1]$.

5.4 Illustration of Step 2.

Based on Assumption 1, suppose that in Lamport's logical clocks, any message sent in region x will be received or lost before region $x+36$. The value of $cl.m$ is a dependent counter i.e. when the value of $cl.m$ is set, it is set to some current value of free counter (namely, cl value of the sender process). Here we assume that $d=1$ is used while updating the clock at the sender process. Moreover, the value of $cl.j$ will be available for at most 36 additional regions based on the initial assumption. In other words, $cl.m$ is a $\langle 0, 36\rangle$-dependent counter. Hence, the above analysis requires that when a process receives a message m in region r, it checks whether its timestamp is at least $3*(r-38)*max_{inc}$ and at most $3*(r+1)*max_{inc}+2*max_{inc}-1$.

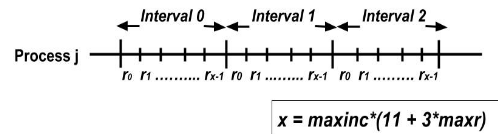

Fig. 2 Determining bounds from Intervals

5.5 Algorithm for Step 3: Bounding the Counters

Steps 1 and 2 focused on relating free and dependent counters to physical time. Recall that if a process is in region r, then any free counter is in the range $[3*r*max_{inc}..3*(r+1)*max_{inc}+2*max_{inc}-1]$ and the value of any dependent counter is in the range $[3*(r-2-max_r)*max_{inc}..3*(r+1)*max_{inc}+2*max_{inc}-1]$. Observe that the size of the above range is $max_{inc}*(11+3*max_r)$. In Step 3, we revise the program so that instead of maintaining each counter to be an unbounded variable, we only maintain it in modulo B arithmetic, where B is 3 times the range of any dependent counter i.e., $B = 3*[max_{inc}*(11+3*max_r)]$.

Next, we give a brief description of why the value of 3*$[max_{inc}*(11+3*max_r)]$ was chosen. Towards this

end, we split 3*[$max_{inc} * (11 + 3 * max_r)$] into three *intervals*, $[0..max_{inc} * (11 + 3 * max_r) - 1], [max_{inc} * (11 + 3 * max_r)..2max_{inc} * (11 + 3 * max_r) - 1]$ and $[2max_{inc}*(11+3*max_r)..3max_{inc}*(11+3*max_r)-1]$. **Each interval corresponds to the range of dependent counters** (cf. Figure 5.4). First observe that the interval is long enough to ensure that all free counters stabilize to their expected values. Regarding dependent counters, consider the case where a process, say j, is about to move from Interval 0 to Interval 1. Since the program can be perturbed to an arbitrary state, at this point, a dependent counter could be in any interval. However, any dependent counter that exists when j is about to move to Interval 1 will be removed from the system before process j moves to Interval 2. (Note that the length of the interval was chosen in order to guarantee this property.)

Now, consider the computation of the program where process j just entered Interval 1 and continues its execution until it enters Interval 2. During this computation, process j will discard all dependent counters in Interval 2 because all valid values for dependent counters in Interval 1 are from Interval 0 or Interval 1. Moreover, given the life-span of dependent counters, any dependent counter generated in Interval 0 will be removed before j enters Interval 2. In other words, when the first process enters Interval 2, all dependent counters are from Interval 1. This property will be preserved for all subsequent intervals.

5.6 Illustration of the Step 3.

Since we consider $cl.m$ as a $\langle 0, 36\rangle$-dependent counter and $max_r = 36$, substituting these in 3*[$max_{inc} * (11 + 3 * max_r)$], we obtain a bound of 3570 for max_{inc} =10. So, instead of maintaining variables $cl.j$ and $cl.m$, we maintain them as $cl.j$ *mod* 3570 and $cl.m$ *mod* 3570. Thus, 12 bits are sufficient to represent $cl.j$ and $cl.m$. Figure 3(a) shows that even in a system with very pessimistic considerations like maximum message delay of about an 1 hour and $max_{inc} = 10^9$ i.e. maximum growth of 10^9 in counters within a region (of 100s, i.e.,$\mathcal{RS}$ =100), we require 41 bits.

6 Proof of Correctness for the Transformation Algorithm

In this section, we show that our transformation algorithm consisting of 3 steps preserves correctness and stabilization property of the original program. Let p be the original program and let p' be the program obtained after all 3 steps. To facilitate the proof we define the notion of *modulo state*.

Definition 14 (modulo B state). Let s be a state, modulo B state of s, where B is an integer, denoted by s^B is obtained by changing each variable x with unbounded domain in s to x *mod* B.

We extend this to *modulo state predicate* and *modulo computation.* For example, $s_0^B, s_1^B \cdots$ is a modulo B computation of p iff $s_0, s_1, \cdots$ is a computation of p and $\forall w : s_w^B$ is the modulo B state of s_w.

Lemma 1 *In the absence of faults, (i.e., starting from a valid initial state), any computation of p' is also a modulo B computation of p, where $B = 3 * [max_{inc} * (11 + 3 * max_r)]$.*

Proof. Observe that the bounds on free counters are derived based on the property that in any global region, the value of the free counter would be in the range $[3 * r * max_{inc}..3 * (r + 1) * max_{inc} + 2 * max_{inc} - 1]$. Hence, starting from an initial state, there would be no need to reset a free counter before/after execution of an action. Likewise, there is no need to reset a dependent counter. The only additional change to free counters is when a process moves from one region to another.

Now, consider a computation of p', say $\langle s_0, s_1, s_2, \cdots\rangle$. Since s_0 is an initial state of p, $\langle s_0\rangle$ is a modulo B computation-prefix of p from an initial state.

Next, assume that $\langle s_0, s_1, \cdots s_j\rangle$ is a modulo computation prefix of p. From the above discussion, (s_j, s_{j+1}) either executes an action of p or it increases a free counter. In the former case, $\langle s_0, s_1, \cdots s_j, s_{j+1}\rangle$ is clearly a modulo B computation of p. In the latter case, $\{s_0, s_1, \cdots s_j, s_{j+1}\}$ is also a modulo B computation of p by the property of free counters, namely that their value can be incremented at anytime.

From the above discussion it follows that $\langle s_0, s_1, s_2, \cdots\rangle$ is a modulo B computation of p, where $B = 3*[max_{inc}* (11 + 3 * max_r)]$. □

Lemma 2 *A computation of p' that starts from an arbitrary state has a suffix that is a modulo B computation of p, where $B = 3 * [max_{inc} * (11 + 3 * max_r)]$.*

Proof. Consider a computation of p'. Due to local correction of free counters, the value of any free counter of a process in region r would be in the range $[3 * r * max_{inc}..3 * (r + 1) * max_{inc} + 2 * max_{inc} - 1]$. Note that even with taking modulo under 3*[$max_{inc} * (11 + 3 * max_r)$], this value can be uniquely identified given the time at that process. As discussed above, we partition 3*[$max_{inc} * (11 + 3 * max_r)$] into three intervals as shown in Figure 5.4. Wlog, let us assume that the current Interval is 0 as determined by the physical time.

As discussed above, if we consider the computation of p' in the Interval 1, when the first process enters Interval 2, all counters are within Interval 1. Hence, the value of any counter can be uniquely determined from the physical time even if it is maintained in modulo 3*$[max_{inc} * (11+3*max_r)]$ arithmetic. Thus, from this point forward, every transition of p' is also a modulo B transition of p. It follows that, every computation of p' has a suffix that is a modulo B computation of p, where $B = 3 * [max_{inc} * (11 + 3 * max_r)]$. □

Theorem 1 *If p is stabilizing to state predicate S then p' is stabilizing to S^B, where $S^B = \{s^B | s \in S$ and s^B is the modulo B state of s $\}$, where $B = 3 * [max_{inc} * (11 + 3 * max_r)]$.*

Proof. To show that p' is stabilizing, we need to show that

1. If p' starts from an arbitrary state then it recovers to its legitimate states. This follows from Lemma 2.
2. If p' starts from a legitimate state then in the absence of faults, it remains in legitimate states forever and it is correct with respect to its specification. This follows from Lemma 1.

□

7 Application of our Algorithm in Katz and Perry Framework [14]

In this section we show that our approach can be applied to any algorithm that can benefit from earlier seminal work by Katz and Perry [14] that focuses on adding stabilization to any program. One issue with that approach is that they need to utilize unbounded counters. We show that our approach can be used to bound counters in those applications. First, we describe the original algorithm briefly in Section 7.1. Then in Section 7.2 we discuss about extending/modifying the original algorithm based on our approach. In Section 7.3 we analyze the number of bits required in the implementation of our algorithm in [14].In Section 7.4 we briefly compare our approach with that in [14].

7.1 Algorithm in [14]

In [14], Katz and Perry presented an algorithm to add stabilization to an existing program. The key idea of the algorithm is as follows: (1) An initiator performs a snapshot of the system using the algorithm by Chandy and Lamport [5]. (2) If the snapshot indicates that the program is not in a legitimate state then it performs a reset whereby the program state is restored to a legitimate state and the computation proceeds thereafter.

To enable calculation of the snapshot and to perform reset without stopping the program execution, the snapshot/reset initiator process utilizes a round number –*which is an unbounded integer variable*. This value is incremented (by the evaluator/initiator process) every time a new reset is performed. Furthermore, if a process in round x receives a message with round y then (1) if $x < y$, the process moves to round y, (2) if $x = y$ then the process treats it as a normal program execution and (3) if $x > y$ then it ignores that message.

7.2 Transforming Algorithm in [14] based on our algorithm

While the round number in [14] does not meet the requirements of the free or dependent counter directly, we can modify the program slightly so that the new variables meet the constraints of free/dependent counter. First, we change the algorithm so that after every snapshot, a reset is performed. However, a Boolean variable is used to identify that this reset is fake and, hence, processes should simply move to the next round but continue accepting previous messages and not perform actual reset. Towards this end, we maintain the following variables: (1) variable nr that is maintained at the initiator only to identify the next round it should use for performing reset, (2) variable cr that is maintained at all processes to identify the current round. This variable is in the same spirit as the one in the algorithm in [14] (3) variable b is a Boolean that identifies whether the reset being done is real or fake. This variable is always 0 except for the moment when the process wants to perform a *real* reset because the snapshot indicated that the state is not legitimate, and (4) variable lr that denotes the sequence number of the last *real* reset performed in the system.

With this change, we can observe that (1) nr is a free counter; it can be increased at will. (2) cr can be viewed as a dependent counter provided we split the action that increases the current round number into $cr = \bot$; and $cr =$ `new value of current round`. In this case, whenever cr is changed, we treat it as if it was first set to $\bot$ thereby removing this dependent counter from the system. Then, we set it to the new value (value of nr) thereby creating a new dependent counter in the system. (3) lr is a dependent counter since lr needs to be set when a new *real* reset is performed. It can be set to $\bot$ after sufficient time to ensure that all existing messages in the system have been delivered.

Remark 9 Note that the above discussion also illustrates that neither the requirement that the dependent counter cannot be changed nor that it is reset to $\perp$ is restricting. Essentially, we need to treat an update of a dependent variable as a two-step process where we first remove the old value of the dependent counter and then initialize it as a new dependent counter (which happens to have the same name) with the new value. All that our definition requires is that the old value is no longer relevant for subsequent computations and that the new value of dependent counter is set to a recent value of some free counter.

7.3 Analysis of bits required for implementation of [14] based on our algorithm

Figure 3(e) shows the size of counters that is sufficient to preserve stabilization provided by [14]. As mentioned in Assumption 1, we consider that the program (implementation of [14]) is running in a $\langle \mathcal{RS}, max_{inc} \rangle$-system with $\mathcal{RS}$ =100s. In such a system, we consider different parameters for message delay on a single channel and number of resets performed in a 100 seconds window to compute the size of counters. We assume that value of any counter increases by at most 1 at each reset. Even if one makes really conservative assumptions like, a message delay of up to one hour and max_{inc} =100 i.e.,100 resets could be performed in a 100 seconds window, the size of the counters is very small (21 bits). The size is even lower for more reasonable parameters. Furthermore, Figure 3(e) shows that the size of counters is not very sensitive to the message delay and number of resets in a window. Therefore, the designer can make very conservative assumptions without increasing the size of counters.

7.4 Comparison of Our Approach with that by Katz and Perry

The solution in [14] is intended to *add stabilization* to an existing program. It involves computation of snapshot and reset (if needed) when snapshot is found to be outside legitimate states. By contrast, our work focuses on *preserving stabilization* of an existing program. It does not rely on snapshot computation to determine whether the current state is legitimate. Therefore, it can be used in algorithms where such snapshot detection is unnecessary. In fact, most stabilizing programs in the literature achieve stabilization without explicitly checking whether the current state is legitimate. For example, the token ring program by Dijkstra [7] allows the system to recover to states where there is a unique token without having any process explicitly check that the current state is illegitimate. Our approach can be used to bound the variables used in such protocols as long as the unbounded variables used in them can be classified in terms of free and dependent counters.

The above discussion showed that our approach can be combined with that in [14]. Given a non-stabilizing program, we can add stabilization via the approach in [14] and then bound the counters involved in it using our algorithm. Additionally, given any stabilizing program with free and dependent counters, our approach can be used to bound those counters. Also, our approach does not require FIFO communication if the original program did not require it.

8 Additional Applications of Our Algorithm

In this section, we demonstrate how our algorithm can be used to transform some more stabilizing programs that use unbounded variables to stabilizing programs that use bounded variables and (practically bounded) physical time. We do so by demonstrating its application in Diffusing Computations, Vector Clocks and Lamport's Mutual Exclusion. In Section 8.1, we show that our algorithm can be applied to diffusing computation which is also applicable to leader election, loop free routing, distributed reset, etc. In Section 8.2, we show that our algorithm can be applied to vector clocks and the resulting algorithm is similar to that in [2]. And, in Section 8.3, we demonstrate that our approach can be used for mutual exclusion algorithm. Also, in this section, we include only an outline of our approach and discuss why the unbounded variables in the given program are either free or dependent counters. We have chosen these applications because they demonstrate the generality of our approach and they also provide several insights into how our algorithm can be applied in a setting where free and dependent counters may not be visible immediately.

8.1 Application of our Algorithm in Diffusing Computation

The problem of diffusing computation [8] is intended to check/modify all processes in a given system. It is used in many applications such as ensuring loop free routing [11], leader election [18, 20], termination detection [8], mutual exclusion [3], and distributed reset [1]. A diffusing computation is initiated by a process and is propagated to its neighbors who in turn forward it to their neighbors. One important requirement of diffusing computation is that a process will have to know

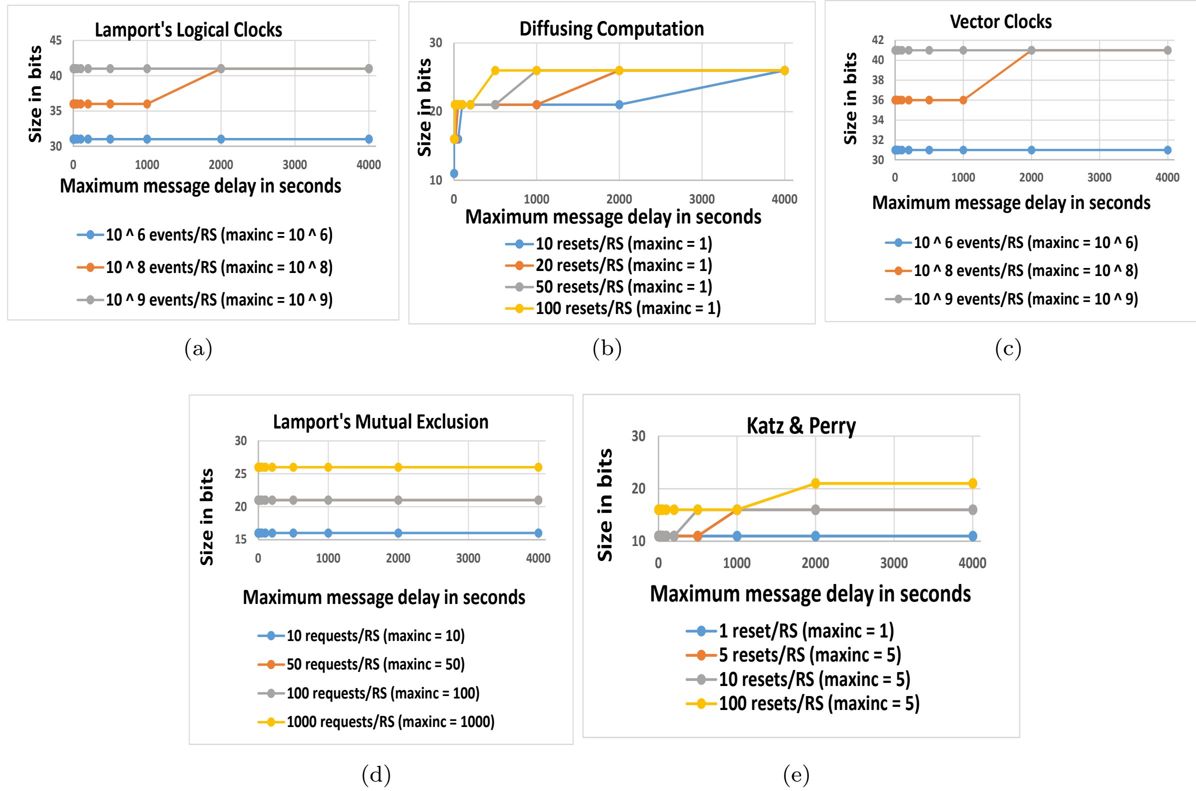

Fig. 3 Analysis of required size in bits for implementation of (a)Lamport's Logical Clocks, (b) Diffusing Computation (no. of processes = 100), (c) Vector Clocks (t=10), (d)Lamport's Logical Clocks based Mutual Exclusion (no. of processes = 100), and (e) [14], with RS = 100s.

whether the diffusing computation that it has received is the same as the one it had received earlier. This can be achieved with an unbounded sequence number that is increased every time a new diffusing computation is initiated. Following our algorithm, we can observe the following:

- The sequence number of the initiator is a free counter; it can be increased by any value when the initiator begins a new diffusing computation.
- The sequence number at other (non-initiator) processes is a dependent counter. It is only relevant when the process begins propagation of the diffusing computation and ends when it completes the diffusing computation. The specific values of (r_b, r_f) for this dependent counter would be determined by the worst case time it would take for a diffusing computation to complete.

Figure 3(b) shows the size of counters needed to achieve this. Once again, even if the input program (diffusing computation) is running in a $\langle \mathcal{RS}, max_{inc} \rangle$-system with $\mathcal{RS}$ =100, max_{inc} =100, i.e., the number of diffusing computations initiated by *one process* in a 100 seconds window is 100 and the value of any counter increases by at most 1 for each computation, and message delay is up to 1 hour, the number of bits required is 26.

8.2 Application of our Algorithm in Vector Clocks

We discussed the application of our algorithm in Lamport's logical clocks in Section 4. We can also extend it to vector clocks [9, 19] or hybrid vector clocks [21]. Vector clocks maintain the variable $vc.j.k$ for each pair of processes j and k. This variable captures the knowledge that j has about k. And, $vc.j.j$ denotes a counter maintained by j for itself. In this program, $vc.j.j$ is a free counter; j can increment the counter maintained by itself by any value it desires. For $j \neq k$, $vc.j.k$ is a dependent counter, provided the underlying communication graph is strongly connected and there exists a time t such that a message (timestamped with vector clocks) is sent on every link in time t. Following this approach, we can obtain a stabilizing program for vector clocks that uses bounded counters. The resulting program is the same as that in [2]. In other words, our

algorithm can be utilized to derive the program in [2]. The size of counters with vector clocks is small (41 bits), as shown in Figure 3(c), even in scenarios where vector clocks is running in a $\langle \mathcal{RS}, max_{inc} \rangle$-system with $\mathcal{RS}$ =100 and max_{inc} =10^9 i.e.,maximum number of events at any process in each window (of size 100 seconds) is 10^9 and value of any counter increases by at most 1 for each event in the system with message delay as long as an hour.

8.3 Application of our Algorithm in Mutual Exclusion

The classic algorithm by Lamport [16] for mutual exclusion utilizes logical clocks (recalled in Section 4). It works as follows: (1) All messages are timestamped with logical timestamps presented in Section 4. (2) When a process wants to access the critical section, it sends a request to all processes. (3) When a process receives the request, it adds the request timestamp to its queue and replies to the requesting process. (4) A process enters critical section iff it has received replies from all processes and if the smallest request contained in its queue corresponds to its own request. And, (5) finally, after a process is done with its critical section, it sends a release message to all other processes thereby allowing others to remove its corresponding request from their queues.

While this algorithm is not typically viewed as a stabilizing algorithm, it can be made stabilizing with simple local checks and corrections. Say, if queue of process j contains a request from k, but process k did not make the request then this request should be removed. The algorithm can ensure stabilization from this state by adding some recovery or corrective actions to handle such scenarios, but it would still involve counters that are unbounded.

In this program, we can observe the following: (1) as discussed in Section 4, the value of $cl.j$, the timestamp of process j is a free counter. (2) Timestamps contained in any message and timestamps saved in the request queue or contained in a request/release message are dependent counters. Following this observation, by applying our transformation algorithm, we can obtain a stabilizing program for mutual exclusion that uses bounded counters. The size of the counters in the transformed program is small (26 bits), as shown in Figure 3(d), even if mutual exclusion is running in a $\langle \mathcal{RS}, max_{inc} \rangle$-system with $\mathcal{RS}$ =100, max_{inc} =1000, i.e., 1000 requests are created in each window (of size 100 seconds) and value of any counter increases by at most 1 for each request in the system with message delay as long as an hour.

9 Effect of clocks differing by multiple regions

As per Definition 10, a program is running in a $\langle \mathcal{RS}, max_{inc} \rangle$ system, if (1) the region of any process differs from the abstract global region by at most 1, and (2) regions of any two processes differ from each other by at most 1 and (3) within a region i.e., within $\mathcal{RS}$ the maximum increase in the value of any variable in the program is max_{inc}. In this section, we consider what happens if we relax the first two properties of this definition. In other words, we would like to generalize the first two properties of Definition 10, so that the region of any process can differ from the abstract global region by at most $nReg$, and regions of any two processes can differ from each other by at most $nReg$. We want to evaluate whether such generalization could help reduce the size of the free counters. Formally, we extend Definition 10 as follows,

Definition 15 We say that a program p is running in a $\langle \mathcal{RS}, max_{inc} \rangle$-system iff for any computation $\langle s_0, s_1, \cdots \rangle$ of p, the following properties are satisfied.

1. At a given state say s_l, for any process j $|\lfloor \frac{t_g(s_l)}{\mathcal{RS}} \rfloor - \lfloor \frac{t_j(s_l)}{\mathcal{RS}} \rfloor| \leq nReg$
2. At a given state say s_l, for any two processes j, k $|\lfloor \frac{t_j(s_l)}{\mathcal{RS}} \rfloor - \lfloor \frac{t_k(s_l)}{\mathcal{RS}} \rfloor| \leq nReg$
3. For any two states say s_a and s_b and any variable $v \in V_p - T_p$, $(\lfloor \frac{t_g(s_a)}{\mathcal{RS}} \rfloor == \lfloor \frac{t_g(s_b)}{\mathcal{RS}} \rfloor) \implies (v(s_b) - v(s_a)) < max_{inc}$

Our transformation algorithm was based on the assumption that $nReg = 1$. For example, let us consider that the program is running in a $\langle \mathcal{RS}, max_{inc} \rangle$ system, where $\mathcal{RS} = t$, $max_{inc} = 10$, $nReg = 1$ i.e., the length of the region is t, the free counter can increase by 10 in one region and regions of two processes differ by at most 1. Equivalently, we *may* be able to consider it as a program running in a $\langle \mathcal{RS}, max_{inc} \rangle$ system, where $\mathcal{RS} = t/2$, $max_{inc} = 5$, $nReg = 2$ i.e., free counters can increase by at most 5 in each region of size $\frac{t}{2}$ if regions associated with processes differ by at-most 2. We want to evaluate whether such an approach could help reduce the size of the free counters. Note that when $nReg$ changes variables max_{inc} and max_r vary accordingly.

Based on this analysis and upon extending our algorithm for $nReg$ regions, if max'_r and max'_{inc} denote the corresponding new max_r and max_{inc} values, we obtain the following new bound B= 3*$[max'_{inc} * (nReg^2 + 5(nReg+1) + max'_r(nReg+2))]$.

After analyzing this new bound on counters with the old bound (comparison discussed in detail in Section A.1 in Appendix) we observed the following:

– The bound on the counters obtained with $nReg > 1$ is smaller than the bound obtained with $nReg = 1$ only for the case where $nReg = 2$. Also, this is true only if the growth in the counters at the processes is distributed uniformly over time and if $max_r = 1$.
– and if $max_r > 1$, then the only beneficial choice is $nReg = 1$, i.e. the notion of modeling region-size such that clocks of any two processes (or physical clock of any process and global clock) differ by at most 1 region.

10 Discussion and Related Work

One of the questions raised by our work is whether the timing properties utilized in our transformation algorithm affect the generality of the algorithm. We note that given the impossibility of solving consensus, leader election [18, 20] and several other interesting problems in asynchronous systems [10], any fault-tolerant solution to these programs must make some reasonable assumptions about the underlying system. Some typical guarantees are process speeds, message delays, etc. Our algorithm utilizes assumptions of this nature to identify free and dependent counters. **Also, as shown in our case studies, even trivially satisfiable requirements –such as clocks differ by at most $100s$ (when current state of the art guarantees synchronization to be less than 10 milliseconds) or maximum growth in counters in a given region is 10^9 or a message is delivered within an hour– are suffice to bound the variables within acceptable limits.**

Not all programs that use unbounded counters can be used with our transformation algorithm. For example, consider algorithms such as those for causal broadcast that maintain an unbounded counter to keep track of number of messages sent by each process. We cannot treat this as a free counter since incrementing it would require us to send broadcast messages. Thus, there are programs where unbounded counters are neither free nor dependent counters.

Our work also differs from previous works that use distributed reset mechanism [1,15] to bound the values of counters. Distributed reset affects all processes. By contrast, stabilization can often be achieved by only processes in the *vicinity* of the affected processes [6,13].

Compared with the work in [4] which assumes the counter size to be equal to the size of integers (32/64 bit in most systems), our approach has the potential to reduce the size of counters. For example, our analysis of Lamport's logical clocks shows a bound of 3570 is sufficient for its counters. In other words, bound depends upon the need of the given application. Also, algorithm in [4] requires multiple/all processes to reset their counters if some process has to reset its counters. By contrast, our algorithm, if applied in the context of Paxos, would address this issue by ignoring messages and resetting processes whose counters are affected rather than affecting all processes. Thus, if perturbation is small, it is anticipated that our solution will affect only corrupted processes.

11 Conclusion and Future Work

Our work addresses a key conflict in the context of stabilization: use of unbounded variables in stabilizing programs should be avoided since any implementation of that stabilizing program would rely on allocating large enough but bounded memory to each variable and transient faults could perturb the program to a state where the large bound associated with the variable is reached. Our work uses the observation that (practically bounded) physical time is used in many systems because corruption associated with time is typically easily detectable and correctable. We provide an alternate approach for providing *practically* bounded-space stabilization by utilizing system and application properties such as clock synchronization properties, message delivery properties, etc. Since a rich class of problems easily admit unbounded state-space solutions, our approach can be used to provide solutions where all program variables are bounded.

We demonstrated that our algorithm is applicable in several classic problems in distributed computing, namely logical clocks, mutual exclusion, vector clocks, and diffusing computations. This work also demonstrates that for a rich class of programs, the approach taken by non-stabilizing programs to deal with unbounded variables –provide large enough but bounded space– is feasible even with stabilizing programs.

References

1. Arora, A., Gouda, M.G.: Distributed reset. IEEE Trans. Computers **43**(9), 1026–1038 (1994)
2. Arora, A., Kulkarni, S., Demirbas, M.: Resettable vector clocks. In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 269–278. ACM, NY, USA (2000). DOI 10.1145/343477.343628. URL `http://doi.acm.org/10.1145/343477.343628`
3. Arora, A., Kulkarni, S.S.: Designing masking fault-tolerance via nonmasking fault-tolerance. IEEE Trans. Software Eng. **24**(6), 435–450 (1998). DOI 10.1109/32.689401. URL `http://dx.doi.org/10.1109/32.689401`
4. Blanchard, P., Dolev, S., Beauquier, J., Delaët, S.: Practically Self-stabilizing Paxos Replicated State-Machine, pp. 99–121. Springer International Publishing, Cham

(2014). DOI 10.1007/978-3-319-09581-3_8. URL http://dx.doi.org/10.1007/978-3-319-09581-3_8

5. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1), 63–75 (1985). DOI 10.1145/214451.214456. URL http://doi.acm.org/10.1145/214451.214456
6. Dasgupta, A., Ghosh, S., Xiao, X.: Probabilistic fault-containment. In: T. Masuzawa, S. Tixeuil (eds.) Stabilization, Safety, and Security of Distributed Systems, 9th International Symposium, 2007, Paris, France, November 14-16, 2007, Proceedings, *Lecture Notes in Computer Science*, vol. 4838, pp. 189–203. Springer (2007). DOI 10.1007/978-3-540-76627-8_16. URL http://dx.doi.org/10.1007/978-3-540-76627-8_16
7. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11), 643–644 (1974)
8. Dijkstra, E.W., Scholten, C.S.: Termination detection for diffusing computations. Inf. Process. Lett. **11**(1), 1–4 (1980)
9. Fidge, C.J.: Timestamps in message-passing systems that preserve the partial ordering. Proceedings of the 11th Australian Computer Science Conference **10**(1), 5666 (1988)
10. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985). DOI 10.1145/3149.214121. URL http://doi.acm.org/10.1145/3149.214121
11. Garcia-Luna-Aceves, J.J.: Loop-free routing using diffusing computations. IEEE/ACM Trans. Netw. **1**(1), 130–141 (1993). DOI 10.1109/90.222913. URL http://dx.doi.org/10.1109/90.222913
12. Ghosh, S.: Distributed systems: an algorithmic approach (2014)
13. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault-containing self-stabilizing distributed protocols. Distributed Computing **20**(1), 53–73 (2007). DOI 10.1007/s00446-007-0032-2. URL http://dx.doi.org/10.1007/s00446-007-0032-2
14. Katz, S., Perry, K.J.: Self-stabilizing extensions for meassage-passing systems. Distributed Computing **7**(1), 17–26 (1993). DOI 10.1007/BF02278852. URL http://dx.doi.org/10.1007/BF02278852
15. Kulkarni, S.S., Arora, A.: Multitolerance in distributed reset. Chicago J. Theor. Comput. Sci. (1998). URL http://cjtcs.cs.uchicago.edu/articles/1998/4/contents.html
16. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). DOI 10.1145/359545.359563. URL http://doi.acm.org/10.1145/359545.359563
17. Lamport, L., Lynch, N.A.: Distributed Computing: Models and Methods. MIT Press (1990)
18. Lee, S., Muhammad, R.M., Kim, C.: A Leader Election Algorithm Within Candidates on Ad Hoc Mobile Networks, pp. 728–738. Springer Berlin Heidelberg (2007)
19. Mattern, F.: Virtual time and global states of distributed systems. In: Parallel and Distributed Algorithms, pp. 215–226. North-Holland (1989)
20. Vasudevan, S., Kurose, J.F., Towsley, D.F.: Design and analysis of a leader election algorithm for mobile ad hoc networks. In: 12th IEEE International Conference on Network Protocols, Berlin, Germany, pp. 350–360. IEEE Computer Society (2004). DOI 10.1109/ICNP.2004.1348124. URL http://dx.doi.org/10.1109/ICNP.2004.1348124
21. Yingchareonthawornchai, S., Kulkarni, S.S., Demirbas, M.: Analysis of bounds on hybrid vector clocks. In: OPODIS 2015, December 14-17, 2015, Rennes, France, pp. 34:1–34:17 (2015). DOI 10.4230/LIPIcs.OPODIS.2015.34. URL http://dx.doi.org/10.4230/LIPIcs.OPODIS.2015.34

A Appendix

In this section, we present the detailed analysis of the effect on bound of counters derived by our algorithm when clocks (of processes and global clock) differ from each other by more that one region. This section also includes a summarized table of notations used in this paper.

A.1 Proof of our Claim on the Effect of clocks differing by multiple regions

Our transformation algorithm was based on $nReg = 1$ in Definition 15. For a distributed system of n processes where the physical clocks of the processes are guaranteed to be synchronized within $0.1s$ of each other and w.r.t the global clock, to achieve $nReg = 1$ the designer could choose $\mathcal{RS} = 0.1s$.

In this section, we analyze the effect of varying $nReg$ on the range or size of the counters. In other words, we would like to answer the question "What is the effect on the range of the counters (i.e.,bound on counters determined by the transformation algorithm) if the clocks of processes are more than one region apart (w.r.t each other and w.r.t the global clock) i.e. $nReg > 1$?" Allowing $nReg > 1$ could help the designer to choose a smaller $\mathcal{RS}$. For instance, in the example discussed above, say if the regions identified by the physical clocks of the processes (w.r.t each other and w.r.t global clock) are allowed to differ by at most 100 regions, i.e. $nReg = 100$ then the designer could choose $\mathcal{RS} = 1ms$.

Observe that when $nReg$ changes variables max_{inc} and max_r vary accordingly. For example though the system described above with $nReg = 1$ and $\mathcal{RS} = 0.1s$ can be equivalently modeled with $nReg = 100$ and $\mathcal{RS} = 1ms$, the values of max_r and max_{inc} differ in the two settings. If the growth in the counters is distributed uniformly, and if $max_{inc} = 100$ in the first setting, then max_{inc} would be 1 in the system with the second setting. Similarly, if $max_r = 1$ in the the first setting, then max_r could be 100 in the second setting i.e. with $nReg = 100$ and $\mathcal{RS} = 1ms$.

A.1.1 Bound for multiple regions

If nReg = 1, i.e. clocks of any two processes differ from each other by at most one region and clock of any process differs from the global clock by at most one region, we try to ensure that any free counter is in the range:$[3 * r * max_{inc}..3 * (r + 1) * max_{inc} + 2 * max_{inc} - 1]$ and (derived from the range of the free counters) any dependent counter to be in the range: $[3*(r-2-max_r)*max_{inc}..3*(r+1)*max_{inc}+2*max_{inc}-1]$. The size of this range of dependent counters is: $max_{inc}*(11+3*max_r)$. Based on this size, each unbounded counter in the program is maintained in modulo B arithmetic. In other words, values of unbounded counters of the original program are bounded by B in the transformed program, where

$$B = 3 * [max_{inc} * (11 + 3 * max_r)] \quad (1)$$

If nReg = 2, i.e. clocks of any two processes are allowed to differ from each other by at most 2 regions and clock of any process is allowed to differ from the global clock by at most 2 regions, then we will try to ensure that any free counter is in the range: $[4 * r * max'_{inc} .. 4 * (r + 1) * max'_{inc} + 3 * max'_{inc} - 1]$ and (derived from the range of the free counters) any dependent counter to be in the range: $[4*(r-2-max'_r)*max'_{inc} .. 4 * (r + 1) * max'_{inc} + 3 * max'_{inc} - 1]$.

Recall that when *nReg* changes variables max_{inc} and max_r vary accordingly, this is denoted by variables max'_r and max'_{inc} in the above equations. Now by **generalizing** formulas presented above, i.e. clocks of any two processes are allowed to differ from each other by at most *nReg* regions and clock of any process is allowed to differ from the global clock by at most *nReg* regions, we try to ensure that any free counter is in the range: $[(nReg + 2) * r * max'_{inc}..(nReg + 2) * (r + 1) * max'_{inc} + (nReg + 1) * max'_{inc} - 1]$ and any dependent counter to be in the range: $[(nReg + 2) * (r - (nReg+1) - max_r) * max'_{inc}..(nReg+2)*(r+1)*max'_{inc} + (nReg+1)*max'_{inc} - 1]$. The size of this range of dependent counters is: $max'_{inc}*(nReg^2+5(nReg+1)+max'_r(nReg+2))$. Based on this size, each unbounded counter in the program would be maintained in modulo B arithmetic i.e. values of the unbounded counters of the original program will be bounded by B in the transformed program, where

$$B = 3*[max'_{inc}*(nReg^2+5(nReg+1)+max'_r(nReg+2))] \quad (2)$$

A.1.2 Analyzing bounds for counters when processes differ by multiple regions vs a single region.

Here we analyze if $nReg > 1$ is beneficial over $nReg = 1$, i.e. if the bound (on the counters) identified when $nReg > 1$ is smaller than the bound identified when $nReg = 1$. Formally, if the bound identified with $nReg > 1$ is denoted as *new* (equation (2) in A.1.1) and if the bound identified with $nReg = 1$ is denoted as *old* (equation (1) in A.1.1), then we would like to check if the following is true,

$$old - new \geq 0 \quad (3)$$

$$(3 * [max_{inc} * (11 + 3 * max_r)])- (3*[max'_{inc}*(nReg^2+5(nReg+1)+max'_r(nReg+2))]) \geq 0, \quad t \quad \textit{(where } nReg > 1\textit{)}$$

$$(max_{inc} * (11 + 3 * max_r))- (max'_{inc} * (nReg^2 + 5(nReg + 1) + max'_r(nReg + 2))) \geq 0$$

If the growth in the counters is distributed uniformly across time, then when the region size becomes smaller (or larger) the bound on the growth in counters within a region becomes smaller (and larger respectively). In other words as *nReg* becomes larger, *RS* (region-size) becomes smaller, and max_{inc} (maximum growth in counters within a region) becomes smaller, and if *nReg* becomes smaller, *RS* becomes larger and max_{inc} becomes larger. We apply this notion to the second half of the above equation, i.e.,as *nReg* increases max_{inc} becomes smaller. So the above equation can be rewritten as,

$$(max_{inc} * (11 + 3 * max_r))- (\frac{max_{inc}}{nReg} * (nReg^2 + 5(nReg + 1) + max'_{inc}(nReg + 2)) \geq 0 \quad (4)$$

Also, from Section 5.3 recall that max_r stands for $max(r_b+r_f)$. So as *nReg* increases, max_r also increases. So the above equation becomes,

$$(max_{inc} * (11 + 3 * max_r))- (\frac{max_{inc}}{nReg} * (nReg^2 + 5(nReg+1) + (nReg * max_r)(nReg+2)) \geq 0 \quad (5)$$

Solving the above equation results in the below equation:

$$(6 - nReg - \frac{5}{nReg} + max_r * (1 - nReg)) \geq 0 \quad (6)$$

Here $max_r > 0$, and we will analyze the following two cases (i) $max_r = 1$, (ii) $max_r > 1$.

Substituting (i) in (6), we obtain $nReg \leq 1$ or $nReg \leq 2.5$. So *nReg* has to be 2 for equation (3) to be true. In other words, the bound on the counters when $nReg > 1$ is smaller than the bound obtained with $nReg = 1$ only for the case where $nReg = 2$. Observe that this is true only if the growth in the counters is distributed uniformly over time and if $max_r = 1$.

Substituting (ii) in (6) we obtain that $nReg \leq 1$ or $nReg \leq 1.67$. So we observe that if $max_r > 1$ then the only beneficial choice is $nReg = 1$, i.e. the notion of modeling region-size such that clocks of any two processes (or the physical clock of any process and global clock) differ by at most 1 region.

A.2 Summary of Notations

Generic Variables

p	program
V_p	set of variables of program p
SV_p	dynamic-sized equivalent of V_p, i.e., a dynamically changing collection of only simple variables, obtained by unraveling complex variables of V_p into their constituent simple variables
A_p	set of actions of program p
s	state of program p
s_l	l^{th} state in a computation of program p
$guard$	condition involving variables in V_p
$statement$	task involving update of a subset of variables in V_p
ρ, ρ'	computation prefixes
x	variable in V_p
$x(s)$	value of variable x in state s
fc	free counter
$fc(s_l)$	value of free counter fc in state s_l
w, a, d	positive integers unless specified otherwise
dc	dependent counter
S	set of states
RS	region size
t_g	abstract global time
t_j	physical time at process j
$\lfloor \frac{t_g}{\mathcal{RS}} \rfloor$	abstract global region
$\lfloor \frac{t_j}{\mathcal{RS}} \rfloor$	region of process j
r	region
r_b, r_f	used to characterize *life* of a dependent counter in terms of regions
max_r	maximum of $(r_b + r_f)$ of any dependent counter
max_{inc}	maximum increase in any free counter within a global region
p'	program obtained by applying our transformation algorithm to program p
B	3*$[max_{inc} * (11 + 3 * max_r)]$ or 3 times the range of any dependent counter

Variables in Katz and Perry example

x, y	round numbers
nr	next round
cr	current round
lr	round number when the last real reset was performed
b	boolean variable that identifies if the reset was real or fake

Variables in Lamport's Logical Clocks example

j, k	processes
$cl.j$	logical clock value of process j
m	message
$cl.m$	message timestamp or logical clock value associated with message m
$channel_{j,k}$	complex variable that contains timestamps of messages in transit between process j and process k
v	number of regions within which a message is guaranteed to be delivered at the receiver process

Variables in Vector Clocks example

$vc.j$	vector clock maintained at process j
$vc.j.k$	highest clock or counter value of process k that process j is aware of