Ideal Stabilization

Mikhail Nesterenko
Kent State University, USA

Sébastien Tixeuil
UPMC Sorbonne Universités & IUF, France

Abstract: We propose a new approach to specifying and reasoning about forward recovery fault tolerant programs. We call it *ideal stabilization*. The program is ideally stabilizing if its every state is legitimate. Ideal stabilization allows the specification designer to prescribe, with arbitrary degree of precision, not only the fault-free program behavior but also its recovery operation. Unlike the classic variant, ideal stabilization is particularly suitable for program composition. Specifications may or may not mention all possible states. We identify approaches to designing ideal stabilization to both classes of specifications. For the first class, we state the necessary condition for an ideally stabilizing solution. On the basis of this condition we prove that there is no ideally stabilizing solution to the leader election problem. We illustrate the utility of the concept of ideal stabilization by providing examples of well-known programs and proving them ideally stabilizing. Specifically, we prove ideal stabilization of the conflict manager, the alternator, the propagation of information with feedback and the alternating bit protocol.

Keywords: Distributed algorithms, Fault-tolerance, Self-stabilization, Methodology

Reference to this paper should be made as follows: Mikhail Nesterenko and Sébastien Tixeuil. (2011) 'Ideal Stabilization', *Int. J. Grid and Utility Computing*, Vol. 1, No. 1, pp.1–2.

Biographical notes: Mikhail Nesterenko got his PhD in 1998 from Kansas State University. Presently he is a full professor at Kent State University. He is interested in wireless networking, distributed algorithms and fault-tolerance.

Sébastien Tixeuil is a full professor at the University Pierre & Marie Curie - Paris 6 (France) and Institut Universitaire de France, where he leads the NPA research group. He received his Ph.D. from University of Paris Sud-XI in 2000. His research interests include fault and attack tolerance in dynamic networks and systems.

1 Introduction

A program is *self-stabilizing* [13, 14, 30] (or just *stabilizing*) if, regardless of the initial state, it eventually satisfies its specification. This elegant property enables the program to recover from transient faults or lack of initialization. During this stabilization period the program behavior is unpredictable. It is tempting to try to engineer the specification such that the program behavior during fault-recovery is controlled. For example, the program starts behaving correctly in no more than ten steps, or critical messages are never lost. However, one of the features of classic stabilization is that the program does not have to satisfy the specification for an arbitrary amount of time. That is, the program is free to ignore the recovery constrains built into the specification.

This results in rather limited compositional properties of stabilizing programs. Stabilizing programs are usually composed by layers: the lower level components are not influenced by the higher level components and, after the lower component starts behaving correctly, they higher level, due to stabilization will eventually behave correctly as well. However, if there is non-trivial two-way interaction between components, the stabilization or correct operation of the composed system is not guaranteed. One way of circumventing this shortcoming is stating an additional program property that the solution has to satisfy besides the specification. For example, that the program has to converge to the legitimate state in finite time. In fact, stabilization itself is a program property that cannot be expressed in the specification alone. However, this haphazard approach leads to difficulties in program composition and reasoning about multiple component programs. For example, it is unclear how the program components should be composed in case their extra-specification properties do not match. These shortcomings diminish

the attractiveness of stabilization as a viable fault-tolerance technique.

In this paper we study the class of programs whose every state satisfies the specification. We call such programs *ideally stabilizing*. Related concepts are occasionally considered by fault-tolerance researchers. However, these approaches are often regarded as theoretical curiosity with a few isolated examples of little practical importance. Our thesis is that the opposite is true. Ideal stabilization retains the advantages of classical stabilization while allowing the engineers and program designers to control the program behavior during fault recovery. Moreover, ideally stabilizing program composition is similar to conventional program composition. This eliminates the need to specify extra-specification program properties. Therefore, the vast array of established program composition techniques can be applied to ideally stabilizing programs. We are thus hopeful that our approach to stabilization makes the general concept of self-stabilization more attractive to fault-tolerance practitioners.

Our contribution. In this paper we study two approaches to ideal stabilization. The approaches depend on the specification type. Specification itself is ideal if it allows (i.e. mentions) all possible states. Specification is not ideal otherwise. Ideal stabilization may be possible to both types of specifications.

Ideal stabilization to non-ideal specification uses the approach we call *state displacement*. The specification implementer provides such mapping from program states to specification states that none of the possible program states map to disallowed specification states. We identify the necessary condition for such specifications to allow ideal stabilization and explain how two well-known programs: conflict manager and the alternator use state displacement to achieve ideal stabilization. By way of contrast, we demonstrate how a simplified leader election specification does not satisfy this condition and hence prohibits ideal stabilization.

Ideal stabilization is possible to ideal specification. In this case, the specification should be such that every possible state is allowed. This lets the engineer specify precisely what behavior, including failure recovery, is expected of the program. An ideally stabilizing program, by definition, has to follow this specification exactly. We state a proposition that demonstrates that such programs are rather common. As an example, we consider the problem specifications for two well-known stabilizing programs: propagation of information with feedback and alternating-bit protocol and provide assertional proofs that the programs ideally stabilize to these specifications. In closing, we discuss the composition of ideally stabilizing programs. The concept of *snap-stabilization* [4, 11] is close to ideal stabilization. We address the relationship between ideal stabilization, snap-stabilization and other related concepts in the related literature section.

2 Model

This section introduces the notation and terms we use in the rest of the paper. To the person familiar with the literature on self-stabilization, our notation may look fairly conventional. However, we encourage even the specialists to read this section as the understanding of the results in the further sections hinges on the notions defined in this one.

Program. A *program* consists of a set of N processes. Each process contains a set of variables. Every variable ranges over a fixed domain of values. Variable v of process p is denoted $v.p$. A process *state* is an assignment of a value from its domain to each variable of the process. A program state, in turn, is an assignment of a value to every variable of each process. The Cartesian product of the values of all program variables is program *state space*. That is, the state space defines all states that the program can assume.

Each process also contains a set of actions. An action has the form $\langle name \rangle : \langle guard \rangle \longrightarrow \langle command \rangle$. A *guard* is a predicate over the variables of the process. A *command* is a sequence of assignment and branching statements.

An action whose guard is **true** in some program state is *enabled* in that state. The execution of an enabled action changes the values of program variables and thus transitions the program from one state to another. *Step* is such a transition. A program *computation* is a maximal sequence of steps. By maximality we mean that the computation is either infinite or it ends in a state where none of the actions are enabled. Note that we do not assume any fairness of action execution for infinite computations.

Communication model, extended state. Processes that share variables are *neighbors*. The communication model determines the type and access method of the variables shared by neighbor processes. For example, in *shared-memory* communication model, the process may mention the variables of the neighbor processes in its actions. That is, the process may read the state of its neighbor processes. The *extended* state of the process is the state of its local variables and the variables that the process can read.

Problem specifications and program sequence mapping. *Problem specification* prescribes the program behavior. This is done by defining the program inputs and outputs through *external variables*. External variables are thus either *input* or *output* variables. Input variables are modified by the environment while the program may only read them. The output variables are updated by the program; they are used to display the results of the program computation. Figure 1 summarizes that a particular node makes use of input, extended state, and produces a particular output.

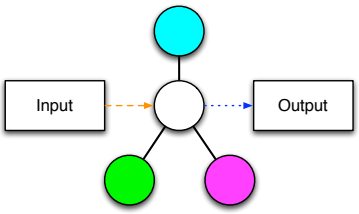


Figure 1 Input, outputs, and the extended state of a process.

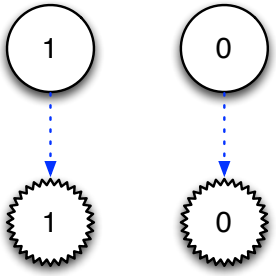


Figure 2 The identity Mapping.

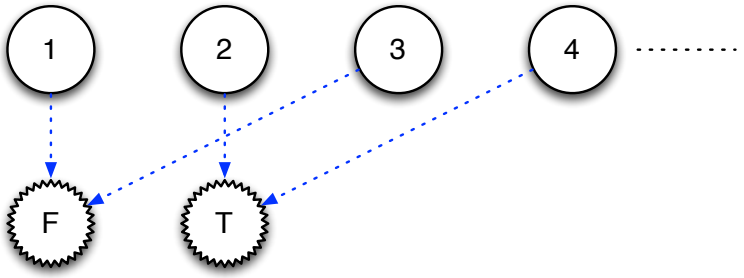
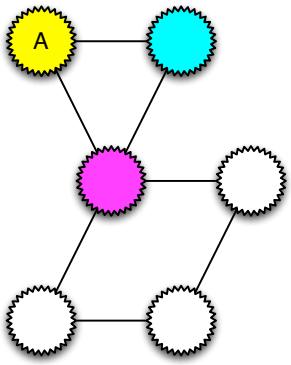
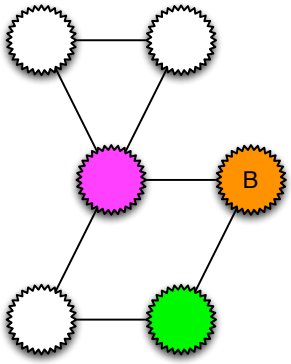


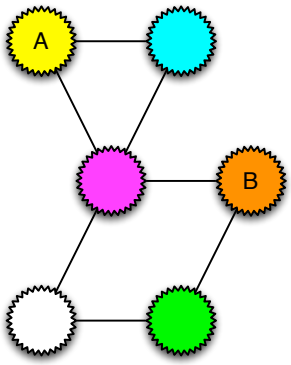
Figure 3 An unambiguous mapping of states.



(a) Mapped state S_1.



(b) Mapped state S_2.



(c) Merged mapped state S_3.

Figure 4 Merge symmetric mapping.

The problem specification is a set of sequences of states of external variables. A program implements the specification. *Input step* in a program computation or a specification sequence is a state transition that updates input variables. A transition that updates output variables is *output step*. Part of the implementation of the specification is the mapping from the program states to the specification states. This mapping does not have to be one-to-one. However, we only consider *unambiguous* programs where each program state maps to only one specification state.

We make another important assumption about program sequence mappings. The mappings have to be *merge-symmetric*. Specifically, let there be a set of program states pr_1 through pr_k such that they map to specification states s_1 through s_k. Then, if there is a specification state s_m such that the extended state of each process p in s_m is the same as in one of the states s_1 through s_k, then there exists a program state pr_m that maps to s_m. In other words, any specification state formed by extended process state-preserving combination of other specification states has a program state that maps to this new specification state. An example of a merge-symmetric mapping is shown in Figures 4(a), 4(b),and 4(c). Ff there exists a mapped state S_1 such that the top-leftmost process maps to state A (Figure 4(a)) and a mapped state S_2 such that the rightmost process maps to state B (see Figure 4(b)), there there also exists a third mapped state S_3 where the two previous processes both map to states A and B, respectively (see Figure 4(c)). If the mapping is not merge-symmetric, then a process may have to differentiate between two global states where this process has exactly the same extended local state. Most known mappings are merge-symmetric.

State mapping is *identical* if the program and the specification use the same state space and every program state maps to the same specification state (See Figure 2). In this case the program uses external variables only.

Another simple program mapping is *projection*. Each process maintains output variables and internal variables for computations and record keeping. The projection of program states onto specification states removes the internal variables. However, the mapping may not be as straightforward as identical mapping or projection. For example, the specification requires the output variable of a process to be boolean while the program maintains an integer variable. The mapping is such that the even values of the integer variable are mapped to **true** while odd values to **false** (See *e.g.* Figure 3).

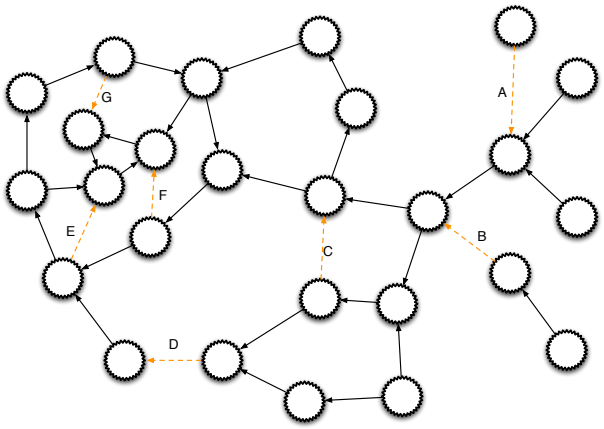
Once the mapping between program and specification states is established, the program computations are mapped to specification sequences as follows. Each program state is mapped to the corresponding specification state. Then, *stuttering*, the consequent identical specification states, is eliminated.

The program does not have to implement all specification sequences. However, the program has to respond to specified input in the manner prescribed by the specification. Informally, the program cannot just ignore "inconvenient" environment input. Hence the following notion. Given a set of sequences $\mathcal{A}$, a subset $\mathcal{B} \subset \mathcal{A}$ is *input-complete* if, for every sequence $\alpha \in \mathcal{A}$, there exists a sequence $\beta \in \mathcal{B}$ such that: every input step s_1 of α is also in β and for every pair of input steps s_1 and s_2 their order in α and β is the same. Informally, the sequences in an input-complete subset $\mathcal{B}$ preserve the results and the order of the input steps in $\mathcal{A}$. Figures 5(a) and 5(b) illustrate this requirement: On Figure 5(b) the full mapping for a given specification is presented, with plain black arrows denoting process actions and dashed orange arrows denoting input actions; by contrast, Figure 5(b) has some mapped states that are not implemented (presented in gray), still all input actions are included by this implementation.

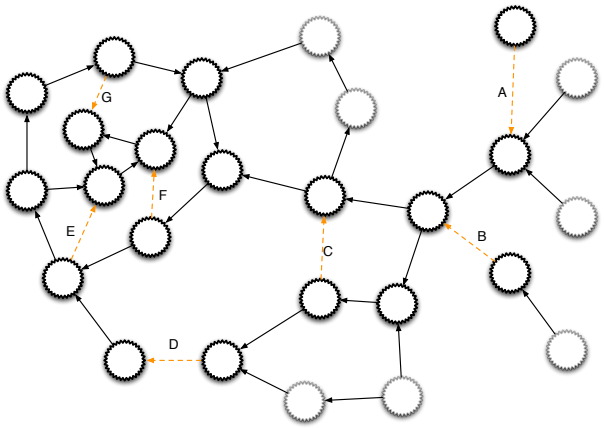
The state space of the specification is the Cartesian product of the ranges of all external variables. The state that is present in one of the specification sequences is *allowed* by the specification. The state is *disallowed* otherwise. The specification is *ideal* if it allows every state in its state space; otherwise the specification is not ideal.

We only consider specifications that are suffix-closed. That is, every suffix of a specification sequence is also a sequence in this specification. Suffix closure enables us to discuss the correctness of the program on the basis of its current state rather than potentially arbitrary long program history. This facilitates assertional reasoning about program correctness.

Predicates, invariants, stabilization. A state *predicate* is a boolean expression over program variables. A program state *conforms* to predicate R, if R evaluates to **true** in this state; otherwise, the state *violates* R. By this definition every state in the program state space conforms to predicate **true** and none conforms to **false**.



(a) Full implementation of a specification.



(b) Limited but input-complete implementation.

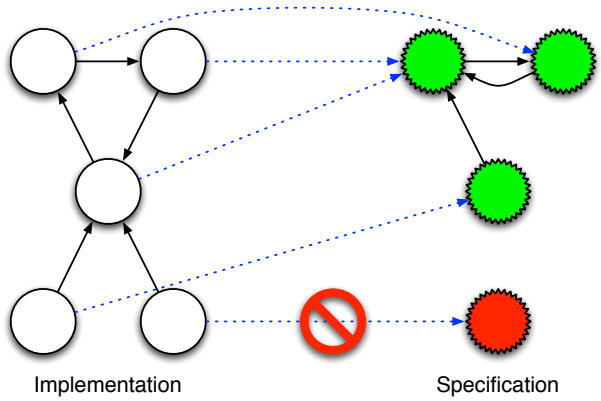
Figure 5 Input complete specification.

The predicate defines a set of program states that conform to it. In the sequel we use the predicate and the set of states it defines interchangeably. Predicate R is *closed* in a certain program $\mathcal{P}$, if every state of every computation of $\mathcal{P}$ conforms to R provided that the computation starts in a state conforming to R.

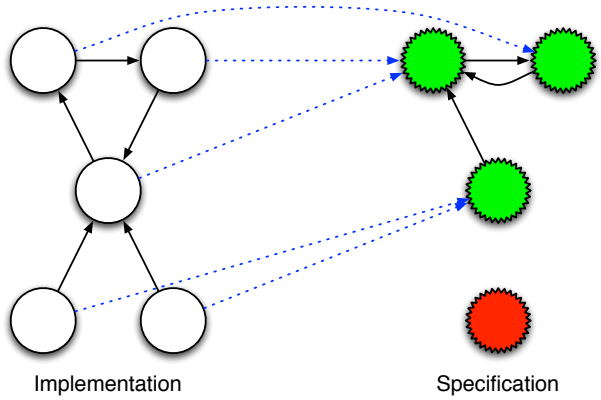
A closed predicate I is an *invariant* of the program $\mathcal{P}$ with respect to specification $\mathcal{S}$ if I has the following property: every computation of $\mathcal{P}$ that starts in a state conforming to I, maps to a sequence that belongs to the specification. A program state is *legitimate* if it conforms to the invariant and *illegitimate* otherwise.

Program $\mathcal{P}$ *satisfies* (or *solves*) specification $\mathcal{S}$, if there exists an invariant I of $\mathcal{P}$ with respect to $\mathcal{S}$ such that the mappings of the program computations that start from I form an input-complete subset of $\mathcal{S}$. That is, the program does not have to implement all the specification sequences, but it does need to accommodate all possible inputs. Specifically, it needs to implement an input-complete subset of these sequences.

A program $\mathcal{P}$ is *stabilizing* to specification $\mathcal{S}$ if every computation that starts in an arbitrary state of the program state space contains a state conforming to the invariant with respect to $\mathcal{S}$. Therefore, any



(a) Non-ideal Specification



(b) State Displacement of the mapped state.

Figure 6 State displacement

computation of a stabilizing program contains a suffix that implements a specification sequence.

Definition 2.1: *A program is* ideally *stabilizing if every state in its state space is legitimate.*

That is, **true** is an invariant of an ideally stabilizing program.

3 State Displacement.

3.1 Necessary Condition for Specification

A non-ideal specification disallows certain states in its state space (see Figure 6(a)). Yet, every state in the program state space is legitimate. Thus, for a program to ideally stabilize to such specification, the state mapping should be such that the disallowed states are displaced. That is, none of the states in the program state space maps to the disallowed states (see Figure 6(b)). However, state displacements may not be possible for an arbitrary specification. The below theorem establishes a necessary condition for a specification to be solvable by an ideally-stabilizing program.

Theorem 1: An ideal stabilization is possible to non-ideal specification only if the specification contains an input-complete subset of sequences such that in every disallowed specification state there is at least one process whose extended state does not occur in any of the specification states of this subset.

Proof: Assume the opposite. There is a non-ideal specification $\mathcal{S}$ that disallows state d and for every input-complete subset $\mathcal{C}$ of $\mathcal{S}$ and for every process p_i, where $i = 1, N$, there is a specification state c_i in one of the sequences of $\mathcal{C}$ such that the extended state of p_i is the same in c_i and in d. However, there is a program $\mathcal{P}$ that ideally stabilizes to $\mathcal{S}$.

Since $\mathcal{P}$ solves $\mathcal{S}$, $\mathcal{P}$ implements an input complete-subset of $\mathcal{S}$. Assume, without loss of generality, that $\mathcal{P}$ implements $\mathcal{C}$. That is, for every sequence of $\mathcal{C}$ there is a computation of $\mathcal{P}$ that maps to this sequence. This means that for each specification state of $\mathcal{C}$, there is a program state of $\mathcal{P}$ that maps to it. This includes the states c_i. For each i, let pr_i be the program state of $\mathcal{P}$ that maps to c_i.

Recall that the extended state of each process in d is the same as in one of the states c_i. If this is the case then, according to merge-symmetry of program mapping, there exists a program state pr_d that maps to d. Since $\mathcal{P}$ ideally stabilizes to $\mathcal{S}$, every state of its state space should be legitimate. That is, every program state has to map to a state in one of the specification sequences. However, the program state pr_d maps to state d which is disallowed by specification $\mathcal{S}$. Which means that pr_d is illegitimate. Hence, contrary to our initial assumption, $\mathcal{P}$ does not ideally stabilize to $\mathcal{S}$. Hence the theorem. □

3.2 Examples

To illustrate the concept of ideal stabilization to non-ideal specifications and the ramifications of Theorem 1, we provide several examples.

Conflict manager. The specification we consider is a simplified (unfair) variant of the dining philosophers problem [7, 12] that we call $\mathcal{UDP}$. The program is adapted from the deterministic conflict manager presented by Gradinariu and Tixeuil [22]. The processes are arranged in a chain. Every process has a unique identifier. The specification defines one external output boolean variable *in* per process. If the value of *in* is **true**, the process may execute the exclusive critical section of code. The specification defines infinite sequences where *in* variables alternate between **true** and **false**. The sequences are not necessarily fair as a certain process may never be given a chance to execute the critical section. That is, an input-complete subset of the specification contains any subset of such sequences.

The specification prohibits concurrent critical section access by neighbor processes. That is, the specification disallows states where *in* variables of two neighbors

are **true** in the same state. Assuming shared memory communication, the extended state of the process contains the state of its neighbors. Hence, none of the allowed specification states contain an extended process state where both the process and one of its neighbors are inside the critical section. The specification thus satisfies the conditions of Theorem 1.

The conflict manager program $\mathcal{CM}$ is as follows. Each process has a single boolean variable *access* and a single action *flip* that is always enabled. The action toggles the value of *access*.

$$\mathit{flip} : \mathbf{true} \longrightarrow \mathit{access} := \neg \mathit{access}$$

The program mapping is this. For each process p the variable $p.in$ is **true** if $p.access$ is **true** and p has the highest identifier among its neighbors with *access* set to **true**.

Let us discuss why this mapping is merge-symmetric. Any extended process state-preserving combination of specification states produces a specification state where neighbors are not accessing the critical section. Then, by appropriately setting *access* variables, we can generate the program state that maps to this specification state.

Let us give an illustration for this reasoning. Assume we have a chain of four processes with identifiers $\langle 2, 1, 3, 4\rangle$. The extended state of each process in this case is its own state plus the state of its left and right neighbors. Consider two specification states

$$s_1 \equiv \langle \mathbf{true}, \mathbf{false}, \mathbf{false}, \mathbf{false}\rangle$$

and

$$s_2 \equiv \langle \mathbf{false}, \mathbf{false}, \mathbf{false}, \mathbf{true}\rangle$$

Some of the program states that map to s_1 and s_2 are respectively $pr_1 \equiv \langle \mathbf{true}, \mathbf{true}, \mathbf{false}, \mathbf{false}\rangle$ and $pr_2 \equiv \langle \mathbf{false}, \mathbf{false}, \mathbf{false}, \mathbf{true}\rangle$.

Specification state $s_3 \equiv \langle \mathbf{true}, \mathbf{false}, \mathbf{false}, \mathbf{true}\rangle$ is formed by merging states s_1 and s_2. Note that the extended states of each process in s_3 are the same in either s_1 or s_2. For example, the extended state of process p_2 is $\langle \mathbf{true}, \mathbf{false}, \mathbf{false}\rangle$, which is the same in s_1 and s_3. Note that there are a number of program states that map to s_3. For example, $pr_3 \equiv \langle \mathbf{true}, \mathbf{false}, \mathbf{true}, \mathbf{true}\rangle$. Thus, the program

Theorem 2: Program $\mathcal{CM}$ ideally stabilizes to the unfair dining philosophers specification $\mathcal{UDP}$.

Proof: To prove ideal stabilization we need to show that a computation of $\mathcal{CM}$ from an arbitrary program state satisfies $\mathcal{UDP}$. First, we show that every state of $\mathcal{CM}$ maps to an allowed state of $\mathcal{UDP}$. Indeed, among the neighbors whose *access* is **true**, *in* is set to **true** only for the process with the highest identifier. That is, every program state maps to the state of $\mathcal{UDP}$ where neighbors do not access the critical section concurrently. Hence, no program state maps to a disallowed state.

Moreover in every computation of the program at least one process, the process with the largest identifier in the system, alternates between setting *access* to **true** and **false**. This means that this process alternates between entering and exiting the critical section indefinitely. Such computations satisfy the specification. That is, $\mathcal{CM}$ ideally stabilizes to $\mathcal{UDP}$. □

Leader election. We present a simplified leader election problem $\mathcal{LE}$ as an example of the specification for which ideally stabilizing solutions do not exist. Again, the processes form a chain. In this case, we only consider $N > 3$. In the external state, each process has two boolean variables: an input variable *contend* and an output variable *leader*. The value of the input variable *contend* is set to a particular value and it does not change throughout the specification sequence. In each specification state *leader* of at most one process is **true**. To exclude trivial solutions, the specification requires that the leader is elected only out of the processes that contend for leadership. That is, the processes whose *contend* variable is **true**. Each specification sequence is finite and ends with a state where the leader is elected. Note that the input complete subset of sequences has to contain a sequence for every combination of the contending processes.

Theorem 3: There does not exist a program that ideally stabilizes to the simplified leader election specification $\mathcal{LE}$.

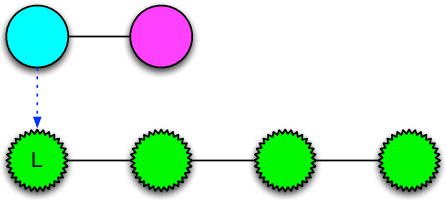
Proof: Let us consider state s_1 (See Figure 7(a)) where the first process is the only one contending for leadership. This is the process that has to be elected leader. That is, the following output state has to be in every input-complete subset. $s_1 \equiv \langle \mathbf{true}, \mathbf{false}, \cdots, \mathbf{false}, \mathbf{false}\rangle$ Similarly, let s_2 be the state where the last process is the only one contending (see Figure 7(b): $s_2 \equiv \langle \mathbf{false}, \mathbf{false}, \cdots, \mathbf{false}, \mathbf{true}\rangle$.

We now form the state s_3 where both the first and the last processes are contending for leadership and both of them are elected (see Figure 7(c)). That is,

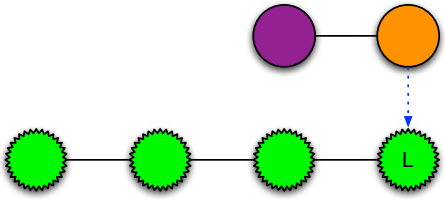
$$s_3 \equiv \langle \mathbf{true}, \mathbf{false}, \cdots, \mathbf{false}, \mathbf{true}\rangle$$

This state is disallowed. Yet, the extended state of every process is present in either s_1 or s_2. Thus, according to Theorem 1, ideal stabilization is not possible to $\mathcal{LE}$. □

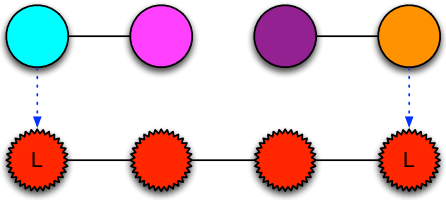
Linear alternator. For another example, we



(a) State s_1



(b) State s_2.



(c) State s_3.

Figure 7 Leader Election

demonstrate how a well-known program called the linear alternator $\mathcal{LA}$ proposed by Gouda and Haddix [20] fits into the definition of ideal stabilization. The alternator provides a solution to the fair variant of the dining philosophers problem $\mathcal{FDP}$. The problem specification is the same as described above except all the sequences are fair with respect to the process critical section access. The modified specification still excludes the states where two neighbors are executing the critical section concurrently and, hence, the specification still satisfies the conditions of Theorem 1. Therefore, the ideal solution is still possible for this specification.

The implementation of $\mathcal{LA}$ is as follows. Similar to $\mathcal{CM}$, each process has a boolean variable x. This time though we assume that the processes in the chain are numbered in the increasing order from 1 to N. The numbering is for presentation purposes only as each process only needs to be aware of its right and left neighbor. The program actions of $\mathcal{LA}$ are shown in Figure 8.

The program-to-specification states mapping is as follows. For each process p_j, the output variable $in.p_j$ evaluates to **true** if process' action is enabled.

Theorem 4: Program $\mathcal{LA}$ ideally stabilizes to the fair dining philosophers specification $\mathcal{FDP}$.

Proof: Gouda and Haddix [20] prove that the alternator satisfies the fairness properties of the dining philosophers specification from an arbitrary program state. We only show the displacement of disallowed

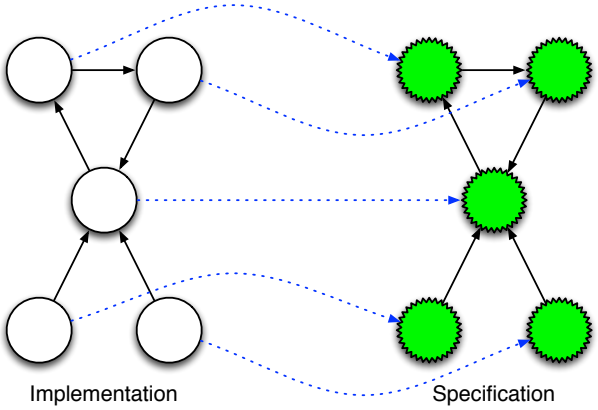


Figure 9 Mapping to an ideal specification.

states. Note that an action of a process p is enabled if $x.p$ is not equal to the left neighbor's variable and equal to the right neighbor's variable. This can only hold for one process in the neighborhood. That is, every program state maps to a specification state where none of the neighbors are in the critical section concurrently. In other words, the program states only map to the allowed states. Hence the theorem. □

4 Stabilizing to Ideal Specifications

4.1 Forming Ideal Specifications

Another method of achieving ideal stabilization is by stating the specification such that all the states in its space are legitimate. That is, stating ideal specification. At first this seems difficult to achieve. However, the following proposition demonstrates that such specifications are rather common.

Proposition 4.1: For every program there is an ideal specification to which this program ideally stabilizes.

We provide an informal argument for the validity of this proposition. Consider any program and all the computations produced by this program when it starts from an arbitrary state of its state space. Now define the specification that contains exactly these computations and identical mapping from program to specification states. This specification is ideal as all the states of its state space are allowed while the program ideally stabilizes to this specification (See Figure 9).

Naturally, this kind of specification may not be very useful as it, in essence, defines the specification to be whatever the program computes. However, below we describe how a number of stabilizing programs published in the literature can be defined as ideally stabilizing to ideal specifications.

4.2 Examples

Propagation of Information with Feedback. As the first example we describe a program $\mathcal{PIF}$ that

$$
\begin{array}{ll}
x.p_1 = x.p_2 & \longrightarrow x.p_1 := \neg x.p_1 \\
(x.p_j \neq x.p_{j-1}) \wedge (x.p_j = x.p_{j+1}) & \longrightarrow x.p_j := \neg x.p_j \\
x.p_{N-1} \neq x.p_N & \longrightarrow x.p_N := \neg x.p_N
\end{array}
$$

Figure 8 $\mathcal{LA}$ program actions. Parameter j ranges from 2 to $N-1$.

ideally stabilizes to the propagation of information with feedback [8, 29] specification. The presentation of this program as snap-stabilizing is well-known [4].

The program is designed for rooted trees. A *root* is an arbitrary distinguished process in the tree. A non-root process with a single neighbor is *leaf*. Processes that are neither roots nor leaves are *intermediate*. For each process u, all processes that lie on the path from u to the root are *ancestors* of u. Process v is a *descendant* of process u, if u is an ancestor of v. Observe that the root is the ancestor of the all other nodes while all of them are the root's descendants. A *parent* of a process is its nearest ancestor. The *height* of a process is the distance to this node's farthest descendant. A *child* of a process is its nearest descendant. A process may have only one parent but many children. A *causality chain* for a leaf is the path of its ancestors to the root. For any two leaves the causality chains have at least one process, the root, in common. Once we are reasoning about a particular causality chain, we assume that it is laid out horizontally with the root located on the left and the leaf on the right.

The tree is organized as follows. Each process has unique identifier throughout the system. For each process, one neighbor's identifier is designated as its parent. This topological designation is constant and incorruptible. If there is no parent, the process is the root. As a shorthand, we assume that the root just has identifier *root*; a leaf's identifier is $leaf$, each process u has set of neighbors $Ch.u$ that are its children and a constant *parent*.

Each process has a state variable *st*. In the intermediate processes, the variable may hold one of the three values: **i**, **rq**, **rp** which stand for *idle*, *requesting* and *replying* respectively. The root can only be idle or requesting while the leaf can be either idle or replying.

The objective of the program is to send a signal from the root to the leaves and, in return, receive an acknowledgment that matches this signal. Operationally, the program should ensure that after the root makes a request then intermediate processes propagate this request in causally ordered steps along each causality chain transitioning from idle to requesting. Afterwards, the intermediate processes propagate the replys from each leaf back to the root in casually ordered steps transitioning from requesting to replying.

We define the following state predicates. For each causality chain whose length is N, $RP(k)$ are the specification states where all k processes on the left are requesting $(k = 1, N-1)$ and the rest of the processes are replying. $RQ(l, m)$ are the specification states where all l processes on the left are requesting $(l = 0, N-1)$ and all $m - l$ processes following them are idle $(m = l+1, N)$, while the remaining processes are replying. The two predicates are mutually exclusive. Specification $\mathcal{SPIF}$ includes the sequences where, for each causality chain, the system satisfies one of the predicates and transitions from one to the other infinitely.

Observe that a step of the root moves the chain from RP to RQ while a step of the leaf moves the chain back from RQ to RP. Since the root is present in each causality chain, the transition from RP to RQ happens in all chains simultaneously, the transition back to RP may differ for each individual chain.

Let us define another pair of predicates. Predicate $RP'(k)$ defines the states where k processes on the left are requesting $(k = 1, N-1)$, the process $k+1$ is replying and the state of the other processes is arbitrary. Notice that $RP(k) \subset RP'(k)$. Predicate $RQ'(l, m)$ are the states where all l processes on the left are requesting $(l = 0, N-1)$, all $m-1$ following them are idle $(m = l+1, N)$ and the state of the other processes is arbitrary. Similarly, $RQ(l, m) \subset RQ'(l, m)$. Specification $\mathcal{IPIF}$ includes the sequences where the system always satisfies either $RP'(k)$ or $RQ'(l, m)$, each sequence has a sequence in $\mathcal{SPIF}$ as a suffix and the transition from $RQ'(l, m)$ is only to $RP(k)$.

We now describe program $\mathcal{PIF}$. It has only external variables. The mapping between the program and specification states is identical. The actions of $\mathcal{PIF}$ are shown in Figure 10.

Lemma 4.2: For any causality chain in the tree, predicate $RQ(l, m) \vee RP(k)$ is closed in $\mathcal{PIF}$.

Proof: (Sketch) Notice that the actions of the processes outside the particular causality chain do not have their variables in the predicate and, therefore, cannot affect it. The closure can then be ascertained by examining the actions of processes in the causality chain. If the program state conforms to $RQ(l, m)$, then the execution of any of the enabled action of a process in this chain moves the system to a state that conforms to either $RQ(l, m)$ or to $RP(k)$. □

Lemma 4.3: For any causality chain, if a computation of $\mathcal{PIF}$ starts in a state conforming to $RQ(l, m)$, this computation also contains a state satisfying $RP(k)$.

Proof: Suppose that initially the program state satisfies $RQ(l, m)$. We show that if $l < N-1$, eventually both l and m are incremented. If $m < N$, *stop* is enabled at process $m+1$ in the causality chain. The execution of this action increments m in $RQ(l, m)$.

$$
\begin{array}{lllll}
\mathit{request}: & & st.root = \mathbf{i} & \wedge (\forall q \in Ch.root : st.q = \mathbf{i}) & \longrightarrow st.root := \mathbf{rq} \\
\mathit{clear}: & & st.root = \mathbf{rq} & \wedge (\forall q \in Ch.root : st.q = \mathbf{rp}) & \longrightarrow st.root := \mathbf{i} \\
\mathit{forward}: & st.parent = \mathbf{rq} & \wedge st.p = \mathbf{i} & \wedge (\forall q \in Ch.p : st.q = \mathbf{i}) & \longrightarrow st.p := \mathbf{rq} \\
\mathit{back}: & st.parent = \mathbf{rq} & \wedge st.p = \mathbf{rq} & \wedge (\forall q \in Ch.p : st.q = \mathbf{rp}) & \longrightarrow st.p := \mathbf{rp} \\
\mathit{stop}: & st.parent = \mathbf{i} & \wedge st.p \neq \mathbf{i} & & \longrightarrow st.p := \mathbf{i} \\
\mathit{reflect}: & st.parent = \mathbf{rq} & \wedge st.leaf = \mathbf{i} & & \longrightarrow st.leaf := \mathbf{rp} \\
\mathit{reset}: & st.parent = \mathbf{i} & \wedge st.leaf = \mathbf{rp} & & \longrightarrow st.leaf := \mathbf{i}
\end{array}
$$

Figure 10 $\mathcal{PIF}$ program actions. Actions *request* and *clear* belong to the root process; actions *forward*, *back*, and *stop* – to an intermediate processes; actions *reflect* and *reset* – to a leaf.

The case of l is a bit more involved. If $l = 0$, the root is idle. The children of the root may or may not be idle. However, if there is a process $q \in Ch.root$ such that $st.q \neq \mathbf{i}$, then *stop* is enabled in q. The execution of this action transitions q to idle. Once all of the root's children are idle, its *request* action is enabled. If it is executed, root transitions to requesting state which increments l. Let us examine the case of $0 < l < N - 1$. If $PQ(l, m)$ is satisfied, process p_{l+1} and its parent are idle. Similar to the case of the root, if p_{l+1} is idle, *stop* is enabled in its each child that is not idle. Once, every child executes *stop*, *forward* becomes enabled in p_{l+1}. If this action is executed, p_{l+1} becomes requesting and l is incremented. That is, if a state of the causality chain satisfies $RQ(l, m)$ both l and m are eventually incremented.

If $l = N - 1$ and $RQ(l, m)$ is satisfied, all processes in the causality chain but the leaf are requesting while the leaf is idle. In this case *reflect* is enabled in the leaf. The execution of this action moves the leaf to the replying state. In this case, the causality chain satisfies $RP(k)$ with $k = N - 1$. That is, the computation that starts in a state that satisfies $RQ(l, m)$ also contains the state satisfying $RP(k)$ □

Lemma 4.4: For any causality chain, if a computation of $\mathcal{PIF}$ starts in a state conforming to $RQ'(l, m)$, this computation also contains a state satisfying $RP(k)$.

The proof of this lemma is similar to the proof of Lemma 4.3.

Lemma 4.5: For any causality chain in the tree, if a computation of $\mathcal{PIF}$ starts in a state conforming to $RP(k)$, it contains a state conforming to $RQ(l, m)$.

Proof: We first demonstrate that if the computation starts in a state where the causality chain satisfies $RP(k)$ with $k > 1$, then it also contains a state where k is decremented. We do it by strong induction on the height of all causality chains in the tree. Specifically, we show that every causality chain of length N reaches a state where the process whose distance from the leaf is $N - k + 1$ is replying. The base case of the distance being zero, i.e. $k = N - 1$, means that only leaf is replying. It follows from Lemma 4.3.

Let us assume that every process at height at least $N - k$ is replying. Let us consider a process p that is not replying and whose height $N - k + 1$. The height of all children of p is at most $N - k$. Due to the assumption, there is a state in this computation where every child of p is replying. Since the chain conforms to $RP(k)$, both p and $parent.p$ are requesting. replying. In this case, *back* action is enabled in p. Once executed, p is replying. By induction this proves our claim.

From this claim, it follows that this computation contains a state where all children of the root process are replying. Since this state conforms to $RQ(k)$, The root is requesting. In this case, action *clear* is enabled in the root process. Once, executed, all causality chains of the system transition to $RQ(0, 1)$. Hence, the lemma. □

Lemma 4.6: For any causality chain in the tree, if a computation of $\mathcal{PIF}$ starts in a state conforming to $RP'(k)$, it contains a state conforming to $RQ(l, m)$

The proof of this lemma is similar to the proof of Lemma 4.5

Theorem 5: $\mathcal{PIF}$ classically stabilizes to $\mathcal{SPIF}$ and ideally stabilizes to $\mathcal{IPIF}$.

Proof: Specification $\mathcal{SPIF}$ requires that for every causality chain in the tree, the solution should remain in the disjunction of the predicates $RQ(l, m)$ and $RP(k)$ and infinitely transition from one to the other. According to Lemma 4.2, the disjunction of predicates is closed in $\mathcal{PIF}$. According to Lemmas 4.3 and 4.5, if the computation starts in a state conforming to one of the predicates, it transitions to the other. Hence, the disjunction of the predicates is the invariant of $\mathcal{PIF}$ with respect to $\mathcal{SPIF}$.

Observe that for every causality chain, the disjunction of predicates $RQ'(l, m) \vee RP'(k)$. That is, it contains the program and specification state space. According to Lemma 4.4, if a computation of $\mathcal{PIF}$ starts in a state conforming to $RQ'(l, m)$, then it also contains a state satisfying $RP(k)$. Similarly, due to Lemma 4.6, if $\mathcal{PIF}$ starts from a state conforming to $RP'(k)$, it transitions to $RQ(l, m)$.This means that the program stabilizes to $\mathcal{SPIF}$ and ideally stabilizes to $\mathcal{IPIF}$. □

Alternating bit protocol. Alternating bit protocol is an elementary data-link network protocol. There is

a number of classic stabilizing implementations of the protocol. Refer to Howell et al [24] for an extensive list of citations. There is also a snap-stabilizing version [11].

The problem is stated as follows. There are two processes: *sender* — p, and *receiver* — q. The processes maintain boolean sequence numbers $ns.p$ and $nr.q$. The processes exchange messages over communication channels. The channels are reliable and their capacity is one. That is, if the channel is empty, the message is reliably sent. If the channel already contains a message, an attempt to send another message leads to the loss of the new message. The processes exchange two types of messages: *data* and *ack*. Both carry the sequence numbers.

The specification $\mathcal{SABP}$ prescribes infinite sequences of states where there is exactly one message in the two channels. The message carries the sequence number of the sender. The state transitions are such that p changes the value of ns. This change is followed by the change of the value in nr that matches the value of ns.

The ideal specification of $\mathcal{IABP}$ is that for every state in the state space there is a sequence that starts in it and every sequence contains a sequence of $\mathcal{SABP}$ as a suffix. Moreover, the non-$\mathcal{SABP}$ prefix of a sequence of $\mathcal{IABP}$ contains no more than five steps.

The program $\mathcal{ABP}$ uses only external variables as described by $\mathcal{SABP}$ and $\mathcal{IABP}$. The mapping from program to specification states is identical. $\mathcal{ABP}$ actions are shown in Figure 11. The sender has two actions: *next* and *timeout*. Action *next* is enabled if there is a message from q in the channel. The timeout action is enabled if there are no messages in either channel. Upon receiving a message from q with matching sequence number, p increments the sequence number and sends the next message. If p times out, it resubmits the same message. The receiver has a single action. When q, receives a message, it sends an acknowledgment back to p. If the message bears a sequence number different from rn, q increments rn signifying the successful receipt of the message.

next: **receive** $ack(nm) \longrightarrow$
  if $nm = ns$ **then**
    $ns := \neg ns$
    send $data(ns)$
timeout: **timeout**() $\longrightarrow$ **send** $data(ns)$
reply: **receive** $data(nm) \longrightarrow$
  if $nm \neq nr$ **then**
    $nr := nm$
  send $ack(nm)$

Figure 11 $\mathcal{ABP}$ actions.

Theorem 6: $\mathcal{ABP}$ classically stabilizes to $\mathcal{SABP}$ and ideally stabilizes to $\mathcal{IABP}$.

Proof: We prove the correctness of the theorem by enumerating the state transitions of $\mathcal{ABP}$. We classify the state space of $\mathcal{ABP}$ into two groups: (i) ns is equal to nr and (ii) ns is not equal to nr. The states are further classified according to the type of messages in the channels. The states and state transitions are shown in Figure 12. Note that to simplify the diagram we do not show the states that contain more than a single message. However, after a single transition, the program moves from one of those states to a state shown in the figure. The correctness of the theorem claims can be ascertained by examining the states and transitions shown in the figure. □

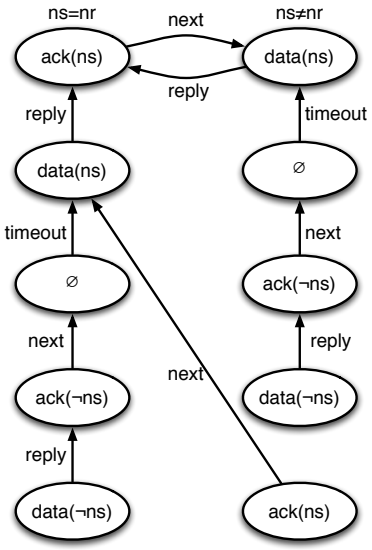


Figure 12 $\mathcal{ABP}$ state transitions.

Note that $\mathcal{ABP}$ can be readily composed with other programs. For example a transport-level protocol that transmits a sequence of messages from the sender to the receiver. Recall that $\mathcal{IABP}$ requires that its solution stabilizes to $\mathcal{SABP}$ in no more than five steps. If that is the case, then $\mathcal{ABP}$ can incorrectly deliver no more than five messages. This tolerance property can be taken into account in composing $\mathcal{ABP}$ with a transport-level protocol.

5 Related Work

We group the related work into three broad categories.

Snap-stabilization. Ideal stabilization is closely related to the notion of snap-stabilization [4, 11]. We regard ideal stabilization as both a restriction and a generalization of snap-stabilization. Snap-stabilization has been defined in two ways: a program is *snap-stabilizing* if its every execution satisfies the specification. In this sense, an ideally stabilizing program is snap-stabilizing to its specification. In fact, some previously published snap-stabilizing programs [25] are ideally stabilizing as their every state is shown to be legitimate. The state displacement technique that we describe in Section 3 demonstrates how to design such

snap-stabilizing protocols. Conversely, ideal stabilization to ideal specifications developed in Section 4 subsumes all known variants of stabilization, including snap-stabilization, as it incorporates arbitrary tolerance properties into the specification itself.

The second definition of snap-stabilization describes a snap-stabilizing program as immediately satisfying an external invocation (such as waves in [9, 10, 11]). Such approach may lead to specifications with sequence-based safety and liveness properties. Proving snap-stabilization to such specifications often results in operational proofs that are difficult to verify. The compositional properties of such programs are unclear. Our definition, on the other hand, allows us to provide assertional correctness proofs for well-known snap-stabilization programs and explore the compositional properties of such programs.

Extra property preserving stabilization. Adding safety properties to self-stabilizing protocols has been an active research direction. However, in order to satisfy a particular safety property, most studies restrict the nature and the extent of the faults. Safe stabilizing algorithms [3, 17] stabilize from an arbitrary initial state and, additionally, can withstand several faults without compromising a certain safety property. Fault-containing stabilization [18] may handle only a single transient failure to ensure actual containment and recovery. Super-stabilizing protocols [15] can withstand one topology change at a time. When faults are simple enough to be detected by checksums [23], it is also possible to maintain elaborate safety predicates. Other approaches to safety enforcement make use of external entities [16] or incorruptible memory [28]. By contrast, ideal stabilization does not restrict the nature or the extent of the transient faults that may affect the system, nor does it require an external safety oracle.

Composition. A number of articles consider composition of classic stabilizing programs [21, 26, 27, 31]. Varghese [31] studies the case where the components are able to stabilize independently. Gouda and Herman [21] discuss adaptive programming where the stabilization of the composed program depends on cooperation of the environment. Leal and Arora [27] provide a detailed study of stabilizing program design by limiting the corruption propagation between components. Candea and Fox [6] organize non-stabilizing program components in a restart tree. If a component gets corrupted, all its descendant components are restarted. There is a string of articles (see, for example, [26]) that explore addition of tolerance components by examining the program state space and eliminating faulty transitions and states. By contrast, the composition of ideally stabilizing programs is as simple as the composition of ordinary programs.

6 New Research Directions

In this paper we proposed a new way of approaching stabilization and its variants. This approach opens a number of new research directions. In conclusion of this paper we would like to outline several proming diretions.

Ideal stabilization legitimizes every possible state thus rendering this term, in conventional sense, meaningless. However, as ideal stabilization shifts emphasis from the implementer to the specification writer, the transitional questions of correct program design are restated differently. Since the ideally stabilizing program satisfies the specification exactly as stated, there is a need to investigate whether there is an equivalent specification-based stabilization to some desired set of states. For example, ideal stabilization may possess a concept that correspond to attractors in pseudo-attractors that by Burns et al [5] defined for self-stabilization. It would be useful to investigate the existence of an equivalent to convergence stair [19] which is a foundation of correctness proofs in self-stabilization as well as local checking and correction [1, 2] which is another well-known proof technique.

Ideal stabilization has interesting compositional properties that make it similar to non-stabilization than to stabilization. Indeed, since a program has to instantly conform to its specification, the composition of two programs is immediate once the two specifications are compatible. Hence, the the composition moves from program to specification. Specification composition has to ensure compatibility during stabilization. For example, if one specification requires recovery from a fault under x steps and another specification requires recovery from a fault in y steps, then the composed specification would have to allow recovery in $x + y$ steps.

A particularly interesting research direction is trying to combine ideal stabilization with tolerance to permanent faults classes such as process crashes, topology changes or Byzantine faults. Unlike transient faults, whose nature is abstracted by considering an arbitrary initial state, permanent faults require changes in the specification of both recovery and post-recovery behavior. A systematic study of the combination of permanent and transient faults would significantly enhance the applicability of ideal stabilization.

References

[1] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[2] B. Awerbuch and G. Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 258–267, 1991.

[3] Alina Bejan, Sukumar Ghosh, and Shrisha Rao. An extended framework of safe stabilization. In David Jeff Jackson, editor, *21st International Conference on Computers and Their Applications, CATA-2006, Seattle, Washington, USA, March 23-25, 2006, Proceedings*, pages 276–282. ISCA, 2006.

[4] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. State-optimal snap-stabilizing PIF in tree networks. In Anish Arora, editor, *1999 ICDCS Workshop on Self-stabilizing Systems, Austin, Texas, June 5, 1999, Proceedings*, pages 78–85. IEEE Computer Society, 1999.

[5] J.E. Burns, M.G. Gouda, and R.E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7:35–42, 1993.

[6] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII*, pages 125–132, May 2001.

[7] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. on Programming Languages and Sys.*, 6(4):632, October 1984.

[8] Ernest J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, 8(4):391–401, July 1982.

[9] A. Cournier, S. Devismes, and V. Villain. From Self-to Snap-Stabilization. In *Stabilization, safety, and security of distributed systems: 8th international symposium, SSS 2006, Dallas, TX, USA, November 17-19, 2006: proceedings*, page 199. Springer-Verlag New York Inc, 2006.

[10] A. Cournier, S. Devismes, and V. Villain. Light enabling snap-stabilization of fundamental protocols. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(1):6, 2009.

[11] Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. Snap-stabilization in message-passing systems. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, page 443. ACM, 2008.

[12] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press N.Y., 1968.

[13] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

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

[15] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago J. Theor. Comput. Sci*, 1997, 1997.

[16] Shlomi Dolev and Frank A. Stomp. Safety assurance via on-line monitoring. *Distributed Computing*, 16(4):269–277, December 2003.

[17] Sukumar Ghosh and Alina Bejan. A framework of safe stabilization. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems, 6th International Symposium, SSS 2003, San Francisco, CA, USA, June 24-25, 2003, Proceedings*, volume 2704 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 2003.

[18] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 45–54, New York, USA, May 1996. ACM.

[19] MG Gouda and N Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

[20] Mohamed G. Gouda and F. Furman Haddix. The alternator. *Distributed Computing*, 20(1):21–28, 2007.

[21] Mohamed G. Gouda and Ted Herman. Adaptive programming. *IEEE Transactions on Software Engineering*, 17(9):911–921, September 1991.

[22] Maria Gradinariu and Sébastien Tixeuil. Conflict managers for self-stabilization without fairness assumption. page 46. IEEE Computer Society, 2007.

[23] Ted Herman and Sriram V. Pemmaraju. Error-detecting codes and fault-containing self-stabilization. *Inf. Process. Lett*, 73(1-2):41–46, 2000.

[24] Rodney R. Howell, Mikhail Nesterenko, and Masaaki Mizuno. Finite-state self-stabilizing protocols in message-passing systems. *J. Parallel Distrib. Comput*, 62(5):792–817, 2002.

[25] Colette Johnen, Luc Alima, Ajoy K. Datta, and Sébastien Tixeuil. Optimal snap-stabilizing neighborhood synchronizer in tree networks. *Parallel Processing Letters*, 12(3-4):327–340, 2002.

[26] Sandeep S. Kulkarni and Anish Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium (FTRTFT 2000) Proceedings*, number 1926 in Lecture Notes in Computer Science, pages 82–93, September 2000.

[27] William Leal and Anish Arora. Scalable self-stabilization via composition. In *24th International Conference on Distributed Computing Systems*, pages 12–21. IEEE Computer Society, March 2004.

[28] Chengdian Lin and Janos Simon. Observing self-stabilization. In Maurice Herlihy, editor, *Proceedings of the 11th Annual Symposium on Principles of Distributed Computing*, pages 113–124, Vancouver, BC, Canada, August 1992. ACM Press.

[29] A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(1):23–35, January 1983.

[30] Sébastien Tixeuil. Self-stabilizing algorithms. In *Algorithms and Theory of Computation Handbook*, chapter 3. Chapma and Hall/CRC Applied Algorithms and Data Structures, 2009.

[31] George Varghese. Compositional proofs of self-stabilizing protocols. In *3rd Workshop on Self-stabilizing Systems, Santa Barbara, California, August, 1997, Proceedings*, pages 80–94. Carleton University Press, 1997.