Stabilizing Pipelines for Streaming Applications

Andrew Berns, Anurag Dasgupta, Sukumar Ghosh
Department of Computer Science
The University of Iowa
Iowa City, Iowa, USA
{andrew-berns, anurag-dasgupta, sukumar-ghosh}@uiowa.edu

***Abstract*—In this paper, we study a compositional approach to designing a class of stabilizing distributed systems. We show that the linear pipelined composition of a number of stabilizing modules is inherently stabilizing, and is a useful method of constructing scalable stabilizing solutions for streaming applications that are on the rise in peer-to-peer and sensor networks. We present the correctness proof and complexity analysis of the composition for a linear pipeline. Subsequently, we generalize the pipelined composition to alternative, concurrent, and repetitive versions, investigate the stabilization properties of these versions, and present a set of conditions under which these extended constructions retain their stabilization properties.**

***Index Terms*—distributed algorithms; distributed computing; fault tolerance;**

I. Introduction

Ad-hoc deployment is a convenient and desirable option for any system. Particularly, for large dynamic distributed systems, initialization on a global scale during deployment is unrealistic. Examples of such systems are peer-to-peer networks and sensor networks. Since the initial state of the system may be unknown, an important property of such systems is their ability to recover from arbitrary initial configurations, and subsequently behave in the expected manner, until a failure or perturbation corrupts its state, and triggers another recovery.

Self-stabilizing systems [1] offer this interesting possibility. A system is *self-stabilizing* when, starting from an arbitrary initial configuration, it eventually recovers to a legal configuration (convergence property), and remains in that configuration thereafter (closure property). Practical applications of stabilizing solutions often stumble around the issue of scale. Proofs that zoom in on actions at the process level in toy-scale systems soon turn out to be impractical when the system is large. Two alternatives have been proposed so far to resolve this problem. The first is the use of reset, which, unfortunately, is more like a sledgehammer, as any fault, no matter how minor, will cause a complete system reset and drastically reduce system availability. The second is the *compositional approach*: given a set of subsystems that are themselves stabilizing, *compose* them into a larger system that is stabilizing by construction. While numerous results are now available for stabilizing toy-scale systems, the success of stabilization research will center around efficient compositional methods, which is in tune with the general approaches to complex system building.

This research was supported in part by NSF grant CNS-0956780

In this paper, we examine a compositional approach to constructing stabilizing solutions for *streaming applications* that have witnessed significant growth in recent years. In P2P systems, audio or video signals are streamed through a large number of client machines [2]. In sensor networks, field data continuously collected by the sensor nodes is routed, filtered, or aggregated on-the-fly as a stream [3]. Location-tracking services (like flight trackers) routinely collect data from position sensors, and route them to remote servers. These applications are characterized by the need to process high-volume data streams in a timely and responsive fashion via a number of machines or communication links that are error prone.

We consider the composition of a number of modules that are known to be stabilizing, in the form of an asynchronous pipeline. The environment supplies an infinite stream of data at the input, which is processed by the pipeline, and sent to the environment as an infinite stream of results at the output. Examples of such system building activities can be seen in wireless sensor networks – the modules here are routers, aggregators, filters and sensor nodes providing customized functionalities. In a linear pipeline, if S_{i-1}, S_i, S_{i+1} are three consecutive modules, then S_i sends an output to S_{i+1} only after the previous module S_{i-1} has produced an output, and the next module S_{i+1} is ready to accept the output from S_i. The readiness of S_{i+1} is conveyed to S_i via some form of acknowledgment. The acknowledgment solves the data overrun problem when the intermediate buffers are full, preventing omission failures en route. Due to ad-hoc deployment, the pipeline may start from an arbitrary initial state, and initially produce meaningless outputs. However, regardless of the starting configuration, in a bounded number of steps, the pipeline is required to produce a meaningful output that reflects a legal behavior. In the rest of the paper, we will use the terms modules, stages, functional units, or subsystems interchangeably to designate the components of the total system.

A. Related Work

Some early work on stabilization with streaming data focused on network protocols. Gouda and Multari [4] studied the stabilization of sliding window protocols, and Afek and Brown [5] presented a randomized protocol for stabilizing the Alternating Bit protocol for the data link layer. Arora et al [6] studied iteration systems that consist of a set of functions on

a set of variables: each enabled function updates a subset of these variables, which potentially enables additional functions in the system. Assuming that these functions belong to different modules, it can be viewed as a study of convergence in a system of interconnected modules. General compositional approaches for stabilization have been studied earlier by [7], [8], [6], [9]. Leal and Arora [7] explicitly identified for each module which other modules it can corrupt, and used appropriate detectors and correctors for these modules to coordinate system correction. Dolev and Herman [8] considered the use of parallelism for stabilization, and proposed a paradigm for composition based on parallel execution of several algorithms and an observer. The most well-accepted technique for composition is layered (also called *vertical*) composition: the system is viewed as a hierarchy of layers, where the interim goals of stabilization for each layer are specified, and the overall system composition progresses in a bottom-up fashion. Gouda and Multari's convergence stairs [4], and Arora and Gouda's distributed reset protocol [10] illustrate this approach. Varghese [9] used the I/O automata model to present a modularity theorem, which is a variation of the layered composition approach. Note that layered composition does not ordinarily address the issue of scale, since each layer typically contains all the processes of the system. However, it untangles the complexity of stabilization proofs. The growing scale of distributed systems handling streaming applications, and the ease of ad-hoc deployment calls for techniques of weaving off-the-shelf stabilizing modules into a complete stabilizing solution, which can be termed a *horizontal* approach. Our work fills this gap.

B. Our contributions

In this paper, we first show that asynchronous pipelines have some inherent stabilizing properties, and then use this result to propose a compositional approach to designing stabilizing distributed systems. This approach, which uses building blocks that are known to be stabilizing, is tailored to streaming applications. Subsequently we generalize the composition by proposing three different types of compositions, and present conditions to be satisfied to make these compositions stabilizing. Finally, we present a general composition theorem for constructing scalable applications with more complex pipelines.

C. The structure of the paper

The paper has five sections. Section 2 introduces the model. Section 3 proves the stabilization properties of linear pipelines. Section 4 generalizes the composition to more complex pipelined structures. Finally, Section 5 contains a few concluding remarks.

II. The model

The system consists of a set V of n processes. A *pipeline* organizes these processes into k functional units or *stages* numbered $1, 2, 3, \ldots, k$ (figure 2). Each stage i is equipped with a buffer B_i: when stage i is ready, it reads the value x from the buffer B_{i-1} of its predecessor $i-1$, computes the function $f_i(x)$, and saves it in B_i. When the successor stage $(i+1)$ is ready, it retrieves the data from B_i, and signals stage i that it has done. At this moment, stage i may accept the next value for processing from its predecessor. We call this composition a *sequential composition*, the pipeline a *linear pipeline*, and denote it by $[1; 2; 3; \cdots k]$.

$\{$*Program for stage* $i : 1 \leq i \leq k$$\}$
do $(v_{i-1} \neq v_i) \wedge (v_{i+1} = v_i) \rightarrow$
$B_i := f_i(B_{i-1}); v_i := \neg v_i;$
od
$\{$*Program for stage* 0*: the source*$\}$
do $v_0 = v_1 \rightarrow$
$v_0 := \neg v_1;$
od
$\{$*Program for stage* $k+1$*: the sink*$\}$
do $v_k \neq v_{k+1} \rightarrow$
$v_{k+1} := v_k;$
od

Fig. 1. Program for Linear Pipeline

Each stage i is composed of a subset V_i of processes such that $\forall i, j : i \neq j :: V_i \cup V_j = \emptyset$. Processes execute actions: each step taken by a process j is a guarded action $g \rightarrow A$, where g is a predicate (involving the state of process j and its neighbors), and A is an action (that updates the state s_i of process i). When g is true for some process in the system, the corresponding action is scheduled for execution. When multiple processes have enabled guards, a weakly fair scheduler chooses a subset of them, and schedules their actions. A *configuration* (also known as a *global state*) is an element of the set $(s_0 \times s_1 \times s_2 \times \cdots \times s_{n-1})$. A *computation* σ is a sequence of configurations – each configuration is reachable from the preceding configuration by executing the actions of a non-empty subset of processes scheduled by the scheduler. Note that the computation of the function f_i by stage i is a *sub-computation* of the system computation that is presumed to be infinite, where a *sub-computation* σ_W for a subset W of processes is obtained from σ by retaining all configurations that contain the processes in W and their immediate neighbors, and removing the rest.

The communication between a pair of neighboring stages $(i, i+1)$ of a linear pipeline is coordinated by a single Boolean variable, v_i, for each process i. Here, $v_i \neq v_{i+1}$ signifies that stage i has valid data in its buffer B_i ready to be consumed by stage $(i+1)$, and $v_i = v_{i+1}$ denotes that stage $(i+1)$ has consumed the previous data from stage i, and is ready to accept the next data. Each stage of a linear pipeline executes the program shown in figure 1. The stages 0 and $(k+1)$ belong to the *environment*. By definition, the environment maintains the predicates $v_0 = \neg v_1$, and $v_{k+1} = v_k$. This means that anytime these conditions are negated, the environment restores them by updating the variables v_0 or v_{k+1}. By doing so, it

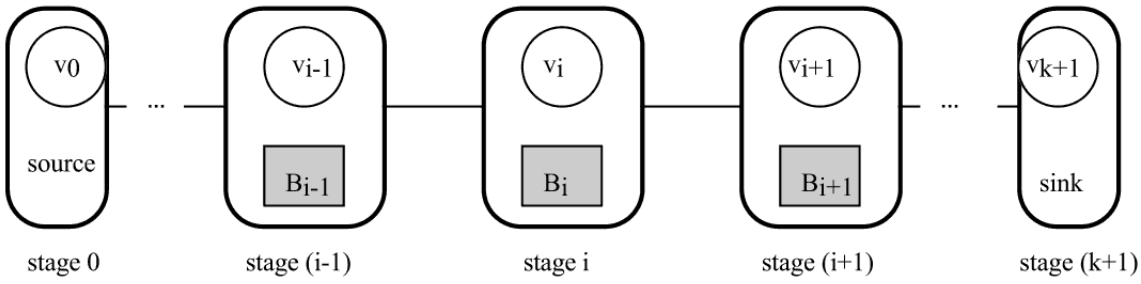

Fig. 2. The structure of a linear pipeline.

simulates an infinite data source at the input (stage 0), and an infinite data sink at the output (stage $(k+1)$) of the pipeline.

Given an input x to stage 1, the pipeline is expected to compute

$$f_k(f_{k-1}(f_{k-2}\cdots(f_2(f_1(x)))))$$

and continue doing so for an infinite stream of input values. However, bad initial configurations may prevent the generation of correct or meaningful outputs. To illustrate this, assume that initially $\forall i : 1 \leq i \leq k, B_i = \lambda$, where λ denotes an arbitrary value. Then $\forall i : 1 \leq i \leq k, f_i(\lambda) = \lambda$. The last stage k will produce a wrong output, if the previous stage $k-1$ incorrectly signals to its successor (by setting $v_{k-1} \neq v_k$) before computing $f_{k-1}(B_{k-2})$ and storing the result in B_{k-1}. This will prompt stage k to read B_{k-1} and compute $f_i(B_{k-1})$, which, by definition is λ. Such observations are relevant to the context of ad-hoc deployment and stabilization.

A. Stage Stabilization

We now present our interpretation of a single stage being stabilizing. We are not concerned about the internal details of a stage, only about its input/output characteristics. Consider that a single stage i is designed to generate the output $f(x)$ for an input x.

Definition 1. *A single input single output pipeline stage is stabilizing when there exists a lower bound m, such that $\forall i \geq m$, given an input x_i, the stage always produces the output $f(x_i)$.*

The above definition can easily be extended to stages with multiple inputs or outputs. Two requirements are relevant here.

Requirement 1. Arbitrary initialization of a single stage should not create a deadlock within that stage.

Requirement 2. The set of variables of each stages includes both *state variables* and the *output variables*. The stabilization of a stage implies that both types of variables return to a legal configuration. Although we are not concerned with the state variables, once the output of a stage stabilizes, within a bounded number of input-output cycles, the state variables must also stabilize. A more exact characterization of these aspects will depend on the internal details of the individual stages.

{Buffer B is a set of timestamped data}
do **true** $\rightarrow$
  receive data m = $(value, src, dst, timestamp)$;
  if $m \notin B \rightarrow$
    add m to B;
    forward m to dst;
  $\square$ $m \in B \rightarrow$ *skip*;
  fi
od

Fig. 3. Program for Data Aggregator

An example of a stabilizing stage is a *data aggregator* that suppresses the transmission of duplicate data packets to the same destination node. An outline of an aggregator is given in figure 3. A corrupted B can lead to the suppression of new messages, or the sending of duplicate messages, but it is easy to observe that such activities will cease to occur after a bounded number of steps.

B. Pipeline Stabilization

A pipeline is stabilizing, when the following holds:

Definition 2. *Let σ_C denote the computation of a correctly initialized pipeline. Then a pipeline is stabilizing when, regardless of the initial configuration, every computation σ contains a suffix that matches σ_C.*

This definition uses the idea of *suffix closure*. An alternative, and more operational definition of a stabilizing pipeline follows:

Definition 3. *A k-stage pipeline is stabilizing if, starting from an arbitrary initial configuration, the system eventually computes the function*

$$f_k(f_{k-1}(f_{k-2}\cdots(f_2(f_1(x)))))$$

for every input x received by it.

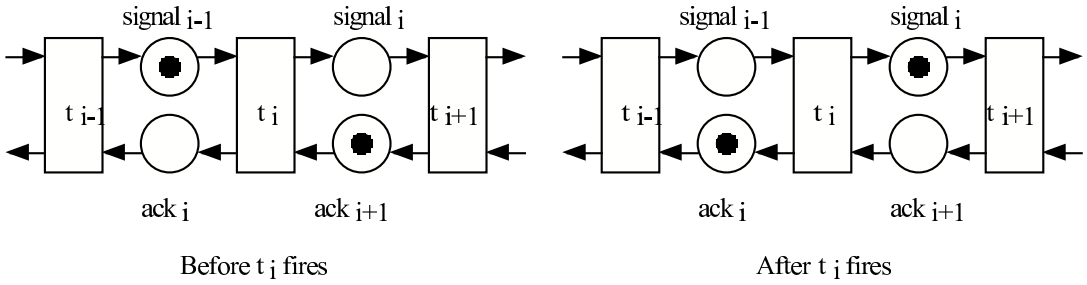

Fig. 4. A Petri net model of a linear pipeline.

In the following section, we demonstrate that every pipeline is *inherently stabilizing*, if the individual stages are stabilizing. By inherent stabilization, we mean that the basic system need not be augmented in any way to guarantee the property of stabilization.

III. Stabilization Property of Linear Pipelines

We abstract a *k-stage* pipeline by constructing a Petri net model of it (figure 4). Each stage i is represented by a single transition t_i with two input places[1]: $signal_{i-1}$ and ack_{i+1}, and two output places: $signal_i$ and ack_i. A *token* in $signal_{i-1}$ indicates that stage $(i-1)$ has computed its function, and stored it in buffer B_{i-1}. Similarly, a *token* in ack_{i+1} implies that stage $(i+1)$ finished computing, and is waiting for the next input from stage i. A transition "fires" i.e. an action is atomically executed, when each of its input places has at least one token. At the end, every input place of that transition loses a token, and every output place gains a token. Figure 4 shows the configurations before and after the firing of transition t_i. After stage i fires, ack_i and $signal_i$ are asserted, as shown by the presence of tokens in those places. This class of Petri nets is called a *marked graph* [11] that has the property that every *place* (denoted by a circle and representing a condition) has at most one input transition and one output transition.

Lemma 1. *There is no deadlock in a linear pipeline.*

Proof: A marked graph is deadlock-free, if every directed cycle has at least one token in it [11]. Note that $signal_{i-1} \equiv v_{i-1} \neq v_i$ and $ack_i \equiv v_{i-1} = v_i$. Therefore, $\forall i : 1 \leq i \leq k+1$, either $signal_{i-1}$ or ack_i must always have one token. Since every directed cycle contains at least one *signal* and one *ack*, the pipeline is deadlock-free. ■

Lemma 2. *In an infinite computation, every transition fires infinitely often.*

Proof: Follows from the property of deadlock-free marked graphs [11]. ■

We now map the legal behavior of a pipeline into a legal computation of the corresponding Petri net model. Observe that between two consecutive firings of a transition t_i ($1 \leq i \leq k$), transitions t_{i-1} and t_{i+1} must fire exactly once. A legal computation of the pipeline starts with the firing of t_1, computing $B_1 = f_1(x)$, followed by a causal chain of transition firings that successively apply the functions f_2 through f_k on it, while accepting new input values of x every time t_1 subsequently fires. When t_1 fires m times, transition t_i ($i > 1$) must fire at least $(m-i+1)$ times, and at most m times. This characterizes σ_C.

Lemma 3. *Starting from any initial configuration, after at most* $\frac{k(k-1)}{2}+1$ *steps, transition* t_1 *has fired at least once.*

Proof: It follows from figure 4 that between two consecutive firings of t_i, transitions t_{i-1} and t_{i+1} must fire exactly once. Assume that in the starting configuration, t_j is an enabled transition that is farthest from the input side. Below, we enumerate for different values of j, the longest firing sequences after which t_1 will fire for the first time.

$j=1$ the empty sequence
$j=2$ $t_2 t_3 t_4 \cdots t_k$
$j=3$ $t_3 t_4 t_5 \cdots t_k t_2 t_3 t_4 \cdots t_k$
$j=4$ $t_4 t_5 \cdots t_k t_3 t_4 t_5 \cdots t_k t_2 t_3 t_4 \cdots t_k$

For $j=1$, the length of the sequence is 0 (excluding the firing of t_1), for $j=2$, it is $(k-1)$, for $j=3$ it is $(k-2)+(k-1)$. Clearly, for $j=k$, the sequence has the largest length of $1+2+3+\cdots(k-1) = \frac{k(k-1)}{2}$. If multiple transitions are enabled initially, then the maximum length of the sequence will not be any longer. Therefore, after at most $\frac{k(k-1)}{2}+1$ steps, transition t_1 will have fired at least once. ■

Theorem 4. *A pipeline is stabilizing, if the individual stages are stabilizing.*

Proof: Since every transition fires infinitely often (Lemma 2), every computation σ has a suffix σ_s that begins with the first firing of t_1. Color all transitions in σ *black*, until t_1 fires for the *first time*, and color the firing of t_1 *red*. Color the firing of any transition that fires causally after t_1, as *red*. In a bounded number of steps, all transitions of the suffix σ_s will be colored red. In the red suffix of the computation, between two consecutive firings of a transition t_i ($1 \leq i \leq k$), transitions t_{i-1} and t_{i+1} will fire exactly once (We do not explicitly include the actions of the environment that simulates the stages t_0 and t_{k+1}). Thus, when t_1 fires m times, each transition t_i ($1 < i < m$) fires at least $(m-i+1)$ times, (and at most

[1]A place is represented by a circle, and a transition is represented by a rectangular box.

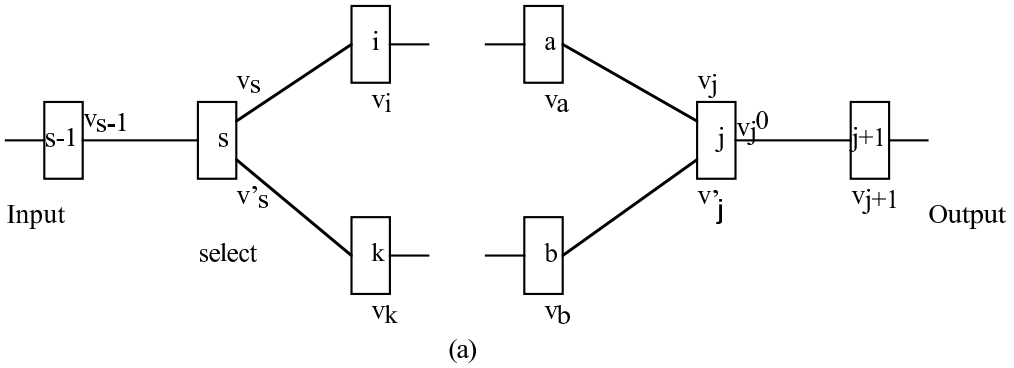

Fig. 5. An example of an eight-stage pipeline based on alternative composition.

m times), which corresponds to the suffix σ_C of the Petri net abstraction of a correctly initialized pipeline.

To show that the original pipeline is stabilizing, recall the interpretation of a stable stage. Starting from an arbitrary initial configuration, assume that a stage i needs a sequence of at most L_i inputs before its output becomes correct. As a result, in the red colored suffix of the computation, a stage i must perform L_i rounds of computation of f_i before its output is certified to be correct and the behavior of the pipeline to be legal. Note that the value of L is always bounded, although it may differ from one stage to another. Eventually, in a sufficiently long finite computation, *every stage* i will compute its output at least L_i times, and the pipeline will stabilize. ■

We now compute an upper bound of the stabilization time for a k-stage pipeline.

Theorem 5. *A linear pipeline consisting of k stages will stabilize after at most $\frac{k(k-1)}{2} + 1 + k(L_{max} - 1)$ time steps, where $L_{max} = max(L_i : 1 \leq i \leq k)$.*

Proof: After at most $\frac{k(k-1)}{2} + 1$ steps, t_1 will have executed at least once (Lemma 3), and all other transitions have fired at least once as well. Afterwards, once a transition fires, it will fire again within at most k time steps. Therefore, after another $k(L_{max}-1)$ time steps, all transitions have fired at least L_{max} times, and the pipeline has stabilized and will produce correct outputs. ■

IV. Extending the topologies

The linear pipeline can easily be extended to other topologies to represent other forms of composition. We consider three specific compositions here: *alternative composition*, *concurrent composition*, and *repetitive composition*.

A. Alternative composition

An alternative composition will be denoted by

$$[1; 2; \ldots s;\ \textbf{if}\ BB\ \textbf{then}\ S1\ \textbf{else}\ S2; j; j+1; \ldots k]$$

where BB is a predicate of stage s, and $S1, S2$ are linear pipelines. Data from stage s is diverted to $S1$ or $S2$ depending on whether the predicate is true or false. Stage j accepts input data from both $S1$ and $S2$ depending on availability, and in an arbitrary order. We call stage s (the stage determining the predicate BB) a *selector* stage, and stage j (the stage where linear pipeline execution resumes) a *join* stage.

{*Program for selector stage s*}
do $BB \wedge (v_{s-1} \neq v_s) \wedge (v_s = v_i) \rightarrow$
$B_s := f_s(B_{s-1})$; $v_s := v_{s-1}$;
□ $\neg BB \wedge (v_{s-1} \neq v'_s) \wedge (v'_s = v_k) \rightarrow$
$B_s := f_s(B_{s-1})$; $v'_s := v_{s-1}$;
od

{*Program for join stage j*}
do $(v^o_j = v_{j+1}) \wedge (v_j \neq v_a) \rightarrow$
$B_j := f_j(B_a)$; $v_j = v_a$; $v^o_j := v_j$;
□ $(v^o_j = v_{j+1}) \wedge (v'_j \neq v_b) \rightarrow$
$B_j := f_j(B_b)$; $v'_j = v_b$; $v^o_j := v'_j$;
od

Fig. 6. Program for Alternative Composition

Figure 5 illustrates an alternative composition with eight stages. Such "select" and "join" operations are quite common in network routing. For example, in a sensor network, data from different sensors eventually joins at the base station. In a peer-to-peer network, the intermediate nodes forward data to different peers depending on the final destination or other characteristics of the data. The selector stage s forwards data to one of two successors i, k and communicates using two boolean variables, (v_s, v'_s). Similarly, the join stage j receives forwarded data from the two predecessor stages $\{a, b\}$, and uses three boolean variables (v_j, v'_j, v^o_j). As with the linear pipeline, when $v_s = v_i$ or $v'_s = v_k$, the selector stage sends the data to the appropriate successor. Similarly, when $v_a \neq v_j$ or $v_b \neq v'_j$, the join stage accepts the input from its predecessor and updates its output flag, v^o_j. The programs for the selector and join stages are given in figure 6.

There are two important observations about the alternative pipeline program.

Lemma 6. *The alternative pipeline program in figure 6 is not stabilizing.*

Proof: There may exist arbitrarily long computations such that BB always evaluates to true (or false), thus excluding the stages starting at k (or i). After any finite prefix of such a computation, a weakly fair scheduler may schedule actions which send output to the dormant stages (in the linear pipeline segment of the lower row of figure 5), which contain arbitrary data in their buffers. Clearly the pipeline can produce bad output at any time. Thus the system is not stabilizing. ■

In order to stabilize an alternative composition pipeline, we need the following guarantee:

Requirement 3. Let (i, k) be the two successors of the select stage s. There must exist an integer $m > 0$, such that within m consecutive executions of stage i, stage k must execute at least once (or vice versa).

Requirement 3 guarantees that in an infinite sequence, all stages (this includes both stage i and stage k) execute infinitely often. After every stage q executes at least L_q times, the suffix of the computation matches σ_C, and the pipeline stabilizes. The stabilization time will depend on m. This leads to the following theorem.

Theorem 7. *If requirement 3 holds, then an alternative composition pipeline with t stages will stabilize after at most $t(t-1)+1+mtL_{max}$ steps.*

Proof: In at most $t(t-1)+1$ steps, stage 1 has executed at least once, and in an additional mt time steps, all stages have executed at least once. From this point, after a stage executes, it must execute again in at most mt time steps. Therefore, in an additional $mt(L_{max}-1)$ time steps, all stages have executed at least L_{max} times. This implies the upper bound on stabilization time is $t(t-1)+1+mtL_{max}$ steps. ■

B. Concurrent composition

Concurrent composition allows a single module to concurrently initiate the computation in multiple modules, or lets one module wait for the completion of multiple modules in the pipeline. We denote a concurrent composition as:

$$[1; 2; 3; \ldots u; S_1||S_2|| \ldots ||S_r; j; j+1; \ldots k]$$

Here, $S_1, S_2, \ldots, S_r$ are linear pipelines. For ease of notation, we assume each such linear pipeline S_i begins with stage s_i^1 and ends with stage s_i^x. Stage u is called a *fork* stage. After stage u completes, each of the pipelines S_1 through S_r initiates their executions, and the join stage j initiates its computation after each of the linear pipelines S_1 through S_r completes their executions. An example of a concurrent composition is given in figure 8. Notice that the use of a single boolean variable per stage (as in a linear pipeline) to coordinate the forking and joining of multiple pipelines leads to deadlock. One can imagine an example where the fork stage u is waiting for pipeline S_i to consume its output, pipeline S_i is waiting for join stage j to consume its input, join stage j is waiting for pipeline stage S_k ($k \neq i$) to produce new output, and pipeline stage S_k is waiting for fork stage u to produce new output. Thus, processes can form a wait-for cycle and deadlock without being able to detect this deadlock using only the boolean variables associated with the stages and pipelines.

{*Program for stages s_i^k ($1 \leq i \leq r$)*}
do $((w_{s_i^{k-1}} \neq w_{s_i^k}) \wedge (w_{s_i^k} = w_{s_i^{k+1}})) \vee$
$\neg(0 \leq (w_{s_i^{k-1}} - w_{s_i^k}) \leq 1) \rightarrow$
$B_{s_i^k} := f_{s_i^k}(B_{s_i^{k-1}})$; $w_{s_i^k} := w_{s_i^{k-1}}$;
od

{*Program for fork stage u*}
do $(v_{u-1} \neq v_u) \wedge$
$(\forall t : 1 \leq t \leq r :: (w_u = w_{s_t^1})) \rightarrow$
$B_u := f_u(B_{u-1})$; $w_u := w_u + 1$;
od

{*Program for join stage j*}
do $((\forall b : 1 \leq b \leq r :: (w_j \neq w_{s_b^x})) \wedge (v_j = v_{j+1}))$
$\vee\neg(\forall b : 1 \leq b \leq r :: w_j = min(w_{s_b^x})) \rightarrow$
$B_j := f_j(B_{s_1^x}, \cdots, B_{s_r^x})$;
$w_j := min(w_{s_1^x}, \cdots, w_{s_r^x})$;
od

Fig. 7. Program for Concurrent Composition with Sequence Numbers

To allow concurrent compositions to be stabilizing, we assign a sequence number w (instead of a single boolean value) to the output of the fork stage. The sequence number is incremented by one each time new data is accepted by the fork stage. The fork stage executes when its predecessor has new data and *all of its successors* have the same sequence number. The stages from the concurrent pipelines $S_1, \ldots, S_r$, as well as the join stage, execute when either the previous stage(s) have processed their input(s), or when a fault condition is detected with the sequence numbers. The modified programs are shown in figure 7.

There are several important observations about the program in figure 7.

Lemma 8. *The concurrent composition given in figure 7 is deadlock-free.*

Proof: First, assume that the pipeline consists only of the concurrent portion - that is, assume that u is the first stage in the pipeline, and stage j is the last stage in the pipeline. Recall that the environment maintains the invariant $v_{u-1} \neq v_u$ and $v_j = v_{j+1}$. We show that, with only a concurrent section, deadlock cannot occur.

To prove this, examine each kind of stage separately. If a sequence number in a stage s_i^1 differs from the sequence number in u, the stage s_i^1 can execute. Similarly, if all stages s_i^x differ from stage j, then stage j has an action to execute. Next, consider the case where the sequence number of some

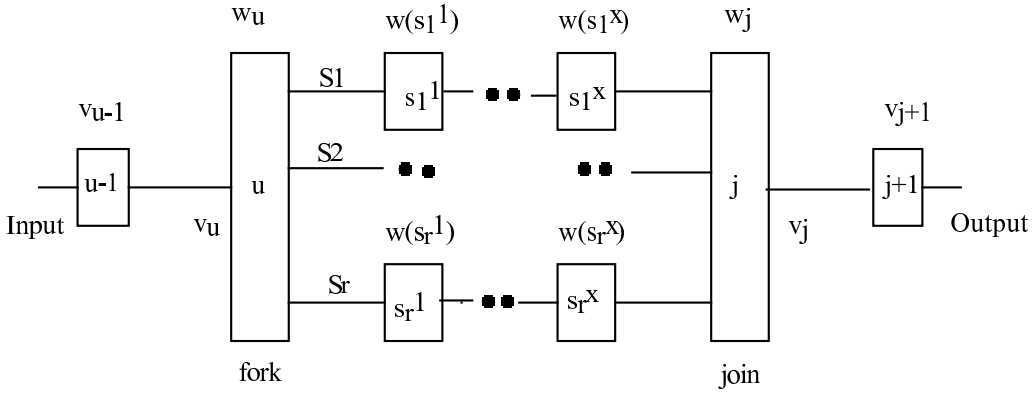

Fig. 8. An example of a pipeline with concurrent composition.

s_i^x is equal to the sequence number of j (w_j), and some are greater than w_j (obviously, if some numbers from stages s_i^x are less than w_j, j has an enabled guard). For all concurrent sections S_d where $w_{s_d^x} = w_j$, the sequence numbers of all stages must be equivalent to w_j to deadlock. This means that stage u has sequence number equivalent to w_j. For all other concurrent sections $S_t, t \neq d$, the sequence numbers must decrease by exactly one from $w_{s_t^1}$ to $w_{s_t^x}$. Since we've assumed $w_{s_t^x} > w_j$, $w_u > w_j$. However, we have shown earlier that unless $w_j = w_u$, some stage will have an enabled guard. This is a contradiction. The remaining case is where all stages have identical sequence numbers. Here, stage u has an enabled guard, and therefore deadlock is impossible in the concurrent section.

Now, we show that in fact $v_{u-1} \neq v_u$ and $v_j = v_{j+1}$ are true infinitely often, even if v_u is not the first stage of the pipeline, and/or v_j is not the last stage of the pipeline. Consider the case where v_{u-1} is the output from a stabilizing pipeline. From the prior pipeline discussion, we can see that a stabilizing pipeline must always produce an infinite stream of outputs (since it is fed with an infinite stream of inputs at one point). Therefore, v_{u-1} will signal new data infinitely often. Similarly, if v_{j+1} is a stabilizing pipeline, it is created with the assumption that it consumes input infinitely often. Therefore, $v_j = v_{j+1}$ will be maintained. Thus, a concurrent system built with stabilizing pipelines before and after the concurrent section still will execute infinitely often (as if interacting with the environment), and so the system is free from deadlock. ■

Lemma 9. *In the concurrent composition program given in figure 7, if one stage executes infinitely often, then all stages execute infinitely often.*

Proof: Observe that each stage, upon execution of an action, can execute only a bounded number of times before both its successor(s) and predecessor(s) have executed. For stage u, executing an action assigns w_u a value not present at any stage s_i^1, and also copies v_{u-1}. This means both stage $u-1$ and stages s_i^1 must execute an action to allow stage u to execute again. Therefore, if stage u executes infinitely often, stage $u-1$ and stages s_i^1 execute infinitely often.

Next, consider a stage s_i^k. After a single execution, $w_{s_i^k} = w_{s_i^{k-1}}$. Stage s_i^k is enabled again only if $w_{s_i^{k-1}} = w_{s_i^{k+1}}$ was true *before* stage s_i^k executed once, or when either stage s_i^{k-1} changes its value such that $\neg(0 \leq w_{s_i^{k-1}} - w_{s_i^k} \leq 1)$, or both stage s_i^{k-1} and stage s_i^{k+1} execute an action. Notice that the first two conditions can only occur when a stage executes an action using the fault guard, $\neg(0 \leq w_{s_i^{k-1}} - w_{s_i^k} \leq 1)$. For stage s_i^1, this guard can be true at most once, as stage w_u will increment solely by one. For stage s_i^2, this guard can be true at most twice. In general, for any stage s_i^l, the fault condition can only be enabled at most l times. Therefore, if a stage s_i^k executes infinitely often, eventually both its successor and predecessor must also execute infinitely often.

Finally, consider stage j. For stage j to execute once, either all stages $s_i^x, 1 \leq i \leq r$ must execute at least once and stage $j+1$ must execute once, or the fault condition $\neg(\forall b : 1 \leq b \leq r :: w_j = min(w_{s_b^x}))$ must evaluate to true. Note that, once j executes, the fault condition cannot be true again until at least one stage s_i^x executes at least once. From the prior paragraph, we know each stage s_i^x can only execute a bounded number of times before its successor and predecessor must also execute. Therefore, the fault condition is only enabled for a bounded number of steps at stage j, at which point all stages $s_i^x, 1 \leq i \leq r$ must execute at least once and stage $j+1$ must execute at least once before stage j can execute again.

Therefore, we have shown that if any one stage executes infinitely often, a subset of other stages must execute infinitely often, which means a larger subset of stages must execute infinitely often, and so on, until all stages must execute infinitely often if at least one does. ■

Theorem 10. *The concurrent composition program given in figure 7 is stabilizing.*

Proof: From lemma 9, we know that if one stage executes infinitely often, all stages execute infinitely often. Furthermore, from lemma 8, we know there is always at least one stage that can execute. Since the system is of finite size, but executes infinitely often, this implies that at least one stage must execute infinitely often, meaning that all stages execute infinitely often.

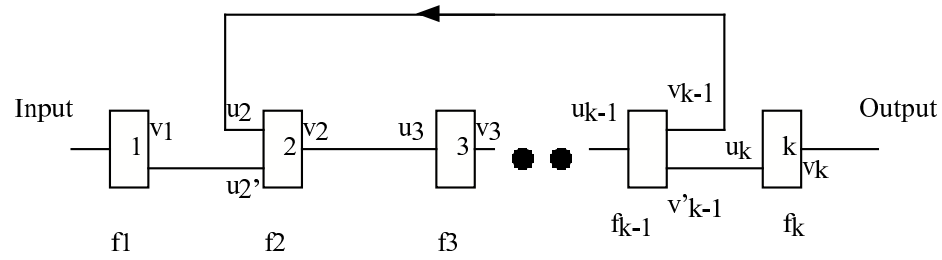

Fig. 9. An example of repetitive composition. Data from stage 4 is fed back to stage 2 when the predicate BB holds for stage 4.

{*Program for loopback stage* p}
if $(v_i \neq v_p) \wedge (v_{s_1} = v_p^o) \rightarrow$
$B_p := f_p(B_i)$;
$v_p^o := \neg v_{s_1}$; $v_p := v_i$;
else if $(v_{p-1} \neq v'_p) \wedge (v_{s_1} = v_p^o) \rightarrow$
$B_p := f_p(B_{p-1})$;
$v_p^o := \neg v_{s_1}$; $v'_p := v_{p-1}$;
fi

{*Program for iterator stage* i}
do $BB \wedge (v_i = v_p) \wedge (v_i \neq v_{i-1}) \rightarrow$
$B_i := f_i(B_{i-1})$;
$v_i = v_{i-1}$; $v_i := \neg v_p$;
$\square$ $\neg BB \wedge (v_i = v_{i+1}) \wedge (v'_i \neq v_{i-1}) \rightarrow$
$B_i := f_i(B_{i-1})$;
$v_i = v_{i-1}$; $v'_i := \neg v_{i-1}$;
od

Fig. 10. Program for Repetitive Composition

Since each stage can execute infinitely often, each stage will eventually stabilize (as each stage was assumed to be a stabilizing pipeline stage). Because each stage will stabilize, the pipeline will eventually compute $f_j(\ldots(x_i)\ldots)$ for all $i > m$, where m is some finite constant. Therefore, the concurrent pipeline using sequence numbers is stabilizing. ■

C. Repetitive composition

The repetitive composition is denoted by

$$[1; 2; \ldots p; \textbf{while } BB \textbf{ do } S; j; j+1; \ldots k]$$

where S is a linear composition. The last stage of S, stage i, is called the iterator stage, while stage p is the loopback stage. Stage i evaluates a predicate BB and, if BB is true, data is routed to stage p, else the data is forwarded to stage j. An example of repetitive composition is illustrated in figure 9. Stage $(k-1)$ is the iterator stage, and stage 2 is the loopback stage. The programs for the iterator i and the loopback p stages are given in figure 10.

Theorem 11. *The repetitive composition program given in figure 10 is stabilizing if there exists an integer $m > 0$, such that in a subcomputation of stage i of length m, data is forwarded at least once to both stage $i+1$ and stage p.*

The proof of this theorem shadows the line of argument used in the alternative composition, and is omitted.

We now state the general composition theorem.

Theorem 12. *If a composition C is stabilizing, and any single stage is substituted by another composition that is also stabilizing, then the resulting composed system C' is also stabilizing.*

The proof can easily be deduced by noting that all stabilizing stages and modules consume infinite inputs and produce infinite outputs. Therefore, substituting one module for another that is also stabilizing will not change the stabilization properties of the overall composition.

Finally, one can generate generalized single-input/single-output pipelines that conform to the following BNF notation.

$< pipe >::=< block > \mid < pipe >< block >$
$< block >::=< stage > \mid < linear > \mid < alternative >$
$\mid < concurrent > \mid < repetitive >$
$< linear >::=< stage > \mid < linear >< stage >$
$< alternative >::=$**if** BB **then** $< block >$ **else** $< block >$ **fi**
$< concurrent >::=< block > \mid\mid < concurrent >$
$< repetitive >::=$**while** BB **do** $< block >$ **end while**

As a consequence of Theorem 12, all such pipelines are stabilizing where the individual stages and the compositions are stabilizing.

V. Conclusion and Future Work

To stabilize the concurrent composition, we assumed that the linear segments have the same size. In case this is not true, the condition can be validated by introducing dummy stages. However, we do not rule out the possibility of the existence of alternative simpler solutions.

A future direction for stabilizing pipelines could be controlling bad outputs. In the current model, a failure in a single stage i will need at most L_i steps before that stage stabilizes and produces correct output. Each faulty output feeds the next stage, and eventually the fault propagates to the final output, meaning a sequence of up to L_i outputs becomes incorrect. When $i = k$, this is unavoidable. However, when $i < k$, a useful exercise is the mending of minor failures, so that it does not affect the final output. Each stage i ($2 \leq i \leq k$) will include a *detector* that detects if the output of the previous stage is consistent with its input. One can debate about the

feasibility of such a detector without a total replication, but sanity checks that lead to a few false negatives may be an acceptable alternative. Each stage i ($1 \leq i \leq k-1$) will send $(x, f_i(x))$ to stage $(i+1)$, where x is the input to stage i. If the check on $(x, f_i(x))$ succeeds, data is forwarded to the next stage – otherwise stage $(i+1)$ will send a negative ack to stage i, demanding a recomputation. The end result is that, following a failure in a single stage, corrupted data will be suppressed before it reaches the final output.

Finally, we recognize the possibility of expanded applications of our work. In addition to sensor and P2P networks, a promising area of application is the stabilization of communicating hardware modules in VLSI, where the system sizes are fast reaching astronomical scales.

REFERENCES

[1] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[2] S. A. Baset and H. G. Schulzrinne, "An analysis of the skype peer-to-peer internet telephony protocol," in *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, April 2006, pp. 1–11.

[3] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A survey on sensor networks," *Communications Magazine, IEEE*, vol. 40, no. 8, pp. 102–114, 2002.

[4] M. G. Gouda and N. J. Multari, "Stabilizing communication protocols," *IEEE Trans. Comput.*, vol. 40, no. 4, pp. 448–458, 1991.

[5] Y. Afek and G. M. Brown, "Self-stabilization of the alternating-bit protocol," in *SRDS*, 1989, pp. 80–83.

[6] A. Arora, P. C. Attie, M. Evangelist, and M. G. Gouda, "Convergence of iteration systems," *Distributed Computing*, vol. 7, no. 1, pp. 43–53, 1993.

[7] W. Leal and A. Arora, "Scalable self-stabilization via composition," in *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 12–21.

[8] S. Dolev and T. Herman, "Parallel composition of stabilizing algorithms," *Distributed Computing Systems, International Conference on*, vol. 0, p. 0025, 1999.

[9] G. Varghese, "Compositional proofs of self-stabilizing protocols." in *WSS*, S. Ghosh and T. Herman, Eds. Carleton University Press, 1997.

[10] A. Arora and M. Gouda, "Distributed reset," *IEEE Trans. Comput.*, vol. 43, no. 9, pp. 1026–1038, 1994.

[11] F. Commoner, A. W. Holt, S. Even, and A. Pnueli, "Marked directed graphs." *Journ. Computer and System Science 5*, pp. 511–523, 1971.