Collaborative Stabilization

Mohammad Roohitavaf and Sandeep Kulkarni
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI
Email: {roohitav, sandeep}@cse.msu.edu

Abstract—In this paper, we present the paradigm of collaborative stabilization that focuses on providing stabilization in the presence of an essential but potentially disruptive environment. By essential, we mean that, without the environment actions, stabilization property would be impossible. At the same time, environment actions are not in the control of the program and can be disruptive to the recovery.

We demonstrate the need for collaborative stabilization by providing examples where existing paradigms of stabilization are undesirable/insufficient. We compare collaborative stabilization with existing paradigms of stabilization. We identify the complexity of verifying collaborative stabilizing programs and develop theorems that focus on composition of such programs.

I. INTRODUCTION

This paper introduces the concept of "Collaborative Stabilization" that focuses on extending the notion of stabilization to work with an essential but potentially disruptive environment, i.e., an environment that is essential for the system to operate but its execution can potentially disrupt the intended execution of the system. To motivate the need for such a concept, consider the following examples:

Shepherding with multiple shepherds. Consider the problem of herding a sheep to reach a desired destination. In this case, we can view the actions taken by (multiple) shepherds to be actions of the program. Actions taken by sheep are the actions of the environment. In this case, actions (movement) of the sheep are essential for convergence of the sheep to the desired location, but at the same time they are potentially disruptive. In other words, the environment actions (movement of the sheep) have the following features:

- They are essential to reach the desired state of the system, and
- they are potentially disruptive, as the sheep may take an action that is not intended by the shepherds.

Therefore, we need to use the potentially disruptive but essential movement of the sheep to reach our desired state.

Pressure cooker. We can view the pressure cooker that is being heated to operate at four pressure levels, Low, Reasonable, High and Very high. It is desirable that the pressure does not exceed the reasonable level for too long. The heat source causes the temperature to increase. Hence, if the heat source increases the temperature from Reasonable to High then the cooker can utilize a whistle mechanism to reduce the temperature to Reasonable. It is undesirable for cooker to stay at High or Very High for a long duration. However, if the pressure is High and the whistle mechanism malfunctions, the program has no mechanism to reduce the temperature to Reasonable. To deal with this scenario, the cooker has another mechanism – a valve – that operates only when the pressure is Very High. This mechanism can only be activated if the heat source continues to increase the pressure from High to Very High. If the heat source does not do this, it is possible for the cooker to stay at High pressure for a long time/forever. And, this is undesirable. When the pressure is Very High, the valve mechanism causes the steam to be released so that the pressure becomes Low. In other words, when the heat source, the environment of this program, increases the pressure from Reasonable to High it was potentially disruptive. However, when it increased from High to Very High, it was essential.

Furnace system. We can view a furnace system to consist of a heater and program that issues commands (turn on/off) to the heater. When the heater is on (respectively, off), it causes the temperature to increase (respectively, decrease). The heater component, however, has some safety features – e.g., preventing it from turning on and off too often. In this case, a heater may continue to stay on even if the program has issued a command to turn off, thereby continue to increase the temperature. In other words, the heater, the environment of this program, is both essential as well as potentially disruptive.

The paradigm of collaborative stabilization proposed in this paper focuses on providing stabilization in the presence of such essential and potentially disruptive environment. The need for such a paradigm is based on the observation that examples such as those presented above are instances of several problems in distributed computing and cyber-physical systems. For example, the shepherding problem can be instantiated into several problems such as (1) a problem of collaborating robots working with live participants in setting such as crowd control, (2) cell migration in a desired direction due to chemical/mechanical signals, (3) forcing a runaway evader (e.g., a runaway plane, boat, criminal) into a desired territory for facilitate its capture and so on. The furnace system is an instance of a cyber-physical system where the computational components rely on potentially non-deterministic physical components. Hence, solutions for it can be generalized to other problems such as a cruise controller operating in the presence of other automobiles and operating on roads with varying slopes and friction, etc.

The paradigm of stabilization [1] cannot be directly applied in a satisfactory fashion to these applications. Stabilization [1] requires that starting from an arbitrary state, after faults stop, the program should recover to its legitimate states. Thus, stabilization implicitly assumes that the perturbation caused by faults is guaranteed to stop (at least for a long enough time to permit recovery). This assumption is not satisfied by the environment actions in the above examples. For example, we cannot assume that the sheep will eventually stop moving in the shepherding program. And, we cannot assume that heat source would be turned off in case of the pressure cooker example.
To deal with non-terminating faults, in [2], authors have proposed the notion of active stabilization. In particular, it focuses on developing a theory of systems that recover in the presence of an active adversary and identifies issues associated with designing, verifying, composing and refining such systems. However, even active stabilization is inappropriate for the problems discussed above. This is due to the fact that unlike adversary actions—which are never essential—environment actions in the above examples are essential. For example, the movement of sheep in the shepherding problem is essential. Hence, if we treat the sheep as adversary and, hence, permit the possibility that the sheep does not move at all then getting the sheep to desired location would be impossible. Likewise, in the pressure cooker, if the heat source does not increase the temperature from High to Very High, it may permit the pressure cooker to stay at High pressure forever. Hence, the valve mechanism would never be activated.

Contributions of the paper.

- We formally define the notion of collaborative stabilization.
- We illustrate the similarity and differences between collaborative stabilization and other variations of stabilizing programs.
- We evaluate the complexity of verifying collaborative stabilizing programs.
- We illustrate how the notion of collaborative stabilization simplifies modeling of programs with an example of a program in the context of shepherding problem.
- We identify how we can create complex collaborative stabilizing programs by composing smaller components.

Organization of the paper. This paper is organized as follows: In Section II, we provide the shepherding problem in more detail as our running example. In Section III, we provide the definition of the program and environment. In Section IV we define the notion of collaborative stabilization. In Section V, as a collaborative stabilizing program, we provide a solution to the shepherding problem. In Section VI, we compare collaborative stabilization with other types of stabilization. In Section VIII, we focus on composition of collaborative stabilizing programs. In Section IX we discuss related work. Finally, in Section X, we conclude the paper and provide the future work.

II. SHEPHERDING PROBLEM

In this section, as a running example to help the reader understand the notions defined in this paper, we define an instance of the shepherding problem introduced in the Section I.

Suppose we have a sheep, a farmer, and a dog moving on a field. The field is represented by a 6 × 6 grid of tiles (see Figure 1). The sheep, farmer, and the dog can move one tile in each step, and they do not move diagonally. Thus, in each move, they can move only one tile to the left, right, up, or down. In each step, either farmer/dog move, or sheep moves. When the sheep moves, before its next move, the farmer and the dog are given a chance to make a move. The sheep cannot go to the border tiles (gray tiles in Figure 1), but the farmer and the dog are allowed to go there.

The farmer and the dog cannot carry the sheep (because it is too heavy!). However, they can steer the sheep by approaching and scaring it. Specifically, the movements of the sheep is as follows:

Movement of the sheep: Sheep scares from the farmer and dog and runs away to increase its distance with them. The sheep computes its distance with each of them by counting the number of steps that are needed for them to reach it.

The problem is to find an algorithm for the farmer and the dog that no matter of the initial location of the farmer, dog and the sheep, sheep always enters the location (1, 4) (marked with star in Figure 1). In one version of the problem, the location (1, 4) is gated, i.e., once the sheep enters it, the sheep cannot leave it. In another version of the problem, the location (1, 4) is not gated, and sheep may leave the location. In the case of ungated location, if the sheep leaves the location, the farmer and dog should steer it back to the location before it leaves the vicinity of the location. Thus, the farmer and dog should not leave the sheep completely unattended.

III. PRELIMINARY

In this section, we define the notions of program and environment. To help the reader understand definitions easier, we use the shepherding problem defined in Section II to explain them.

Definition 1 (Action): An action over the set of variables $V$ is of the form $G ightarrow C$, where $G$ is a predicate (guard) over variables in $V$, and $C$ is a statement that changes the values of (some of) the variables in $V$.

We assume that each variable is associated with a domain and every statement updates the value so that the new value is always in the respective domain. For example, action $a = 0 \rightarrow a := 1$ specifies that if the value of variable $a$ is 0 then it can be changed to 1. Using actions and variables, we define a program as follows:

Definition 2 (Program): A program $p$ is of the form $(V_p, A_p)$ where $V_p$ is a set of variables, and $A_p$ is a set of actions over $V_p$.

Now, consider the above definition in the context of the shepherding problem. We specify the location of the sheep by two variables that represent the column and the row of the sheep. Similarly, we can represent the location of the farmer and dog by a pair of variables for each of them. Thus, the set of program variables, $V_p$, has six different variables as shown in Table I. Note that, although the sheep is not part of the program, s.col and s.row are in program variables, as the
farmer and the dog need to use them to make their decisions.

The actions of the program are the decision made by the farmer and the dog. For example, consider the following action:

\[
(f.\text{row} = 1 \land f.\text{col} = 1) \land (d.\text{row} = 4 \land d.\text{col} = 2) \land \\
(s.\text{row} = 2 \land s.\text{col} = 3)
\]

This action states: if the farmer is in location \((1, 1)\), dog is in location \((4, 2)\), and sheep is in location \((2, 3)\), the farmer moves to location \((1, 2)\) (one tile right), and dog moves to location \((3, 2)\) (one tile up) (see Figure 1).

Similar to program actions, the environment is also specified in terms of a set of actions (that are out of the control of the program), i.e.,

**Definition 3 (Environment):** An environment \(A_e\) for program \(p = \langle V_p, A_p \rangle\) is defined as a set of actions over \(V_p\).

In the shepherding program, the movement of the sheep is considered as the environment. Thus, the environment of this program consists of actions that change \(s.\text{col}\) and \(s.\text{row}\). According to the movement of the sheep define in Section II, we define the environment \(A_e\) as follows:

// Let \((f_r, f_c), (d_r, d_c), (s_r, s_c)\) be locations of farmer, dog and sheep respectively.

\[A_e = \{\text{sheep has a permissible movement} \rightarrow \\
\text{s.row, s.col := s'}_r, s'_c\}
\]

where

\[
\begin{align*}
(C1) &\ |s'_r - s_r| + |s'_c - s_c| = 1 \\
(C2) &\ 0 < s'_r < 5 \land 0 < s'_c < 5 \\
(C3) &\ DIST(f_r, f_c, d_r, d_c, s'_r, s'_c) > \\
&\ DIST(f_r, f_c, d_r, d_c, s_r, s_c))
\end{align*}
\]

In the above actions, the guard specifies all states where there exist \(s'_r\) and \(s'_c\) that satisfy C1, C2, and C3. In addition, in the version of the problem with the gated location \((1, 4)\), sheep does not move in state where \(s_1 = 1\) and \(s_c = 4\).

C1 captures that the sheep can only move one row or one column at a time. C2 captures that the sheep cannot enter the border rows/columns. \(DIST\) represents the distance of the sheep from the farmer and the dog, and it is computed as follows:

\[
DIST(f_r, f_c, d_r, d_c, s_r, s_c) = |s_r - f_r| + |s_c - f_c| + \\
|s_r - d_r| + |s_c - d_c|
\]

C3 captures that sheep tries to increase the distance from the farmer and dog.

The description of program \(p\) includes a set of variables \(V_p\), each of which is associated with a domain. A state of program \(p\) is obtained by assigning each variable a value from its domain. Thus, we define a state and state space of a program as follows:

**Definition 4 (State and State Space):** A state of program \(p = \langle V_p, A_p \rangle\) is obtained by assigning each variable in \(V_p\) a value from its domain. The state space of \(p\) is the set of all possible states.

In the shepherding problem, each state of the program represents the location of the farmer, dog, and sheep. The farmer and dog can be in any of the 36 tiles, but sheep is limited to the 16 inner tiles. Thus, the state space of the program has \(36 \times 36 \times 16\) states.

We also define a state predicate as follows:

**Definition 5 (State Predicate):** A state predicate of \(p\) is a subset of its state space.

As the farmer, dog, and sheep move, the state of the program changes. A change in a state of a program is called a transition. In other words, when the program changes its state from \(s_1\) to \(s_2\) we say program \(p\) transitions from \(s_1\) to \(s_2\). These transitions can be due to the actions of the program or the environment. In fact, each action is a set of program transitions. To illustrate, consider the following example:

Suppose we have a program that has two boolean variables \(a\) and \(b\) each with domain \(\{0, 1\}\). The state space of the program include four states \((0, 0), (0, 1), (1, 0), \) and \((1, 1)\) where the first digit represents the value of \(a\), and the second digit represents the value of \(b\). As we said above, a transition is a change in the state of the program. Thus for example, \((0, 0), (0, 1)\) is a transition for the case where the state of the program changes from \((0, 0)\) to \((0, 1)\).

Now, suppose the program have action \(a = 0 \rightarrow b := 0\) in its set of actions. This action states: if the value of \(a\) is 0, we set \(b\) to 0. Thus, this action is actually the set of transitions \(\{(0, 0), (0, 0)\}, \{(0, 1), (0, 0)\}\).

We formally define the set of program transitions for a program as follows:

**Definition 6 (Program Transitions):** The set of transitions of program \(p = \langle V_p, A_p \rangle\) with state space \(S_p\), is:

\[
\delta_p = \{(s_0, s_1) | s_0, s_1 \in S_p \land \exists G \rightarrow C : G \rightarrow C \in A_p : \langle G \text{ is true in state } s_0 \rangle \land \langle s_1 \text{ is obtained by applying statement } C \text{ in state } s_0 \rangle\}
\]

Similarly, we can define the set of environment transitions as follows:

**Definition 7 (Environment Transitions):** Let \(A_e\) be an environment for program \(p = \langle V_p, A_p \rangle\). The set of transitions of \(A_e\) is:

\[
\delta_e = \{(s_0, s_1) | s_0, s_1 \in S_p \land \exists G \rightarrow C : G \rightarrow C \in A_e : \langle G \text{ is true in state } s_0 \rangle \land \langle s_1 \text{ is obtained by applying statement } C \text{ in state } s_0 \rangle\}
\]

For brevity, we use environment and transitions of the environment interchangeably, i.e., when we say environment \(\delta_e\) we mean an environment with set of transitions \(\delta_e\).

**IV. COLLABORATIVE STABILIZATION**

Using the definitions of the program and environment in Section III, in this section, we define the notion of collaborative stabilization. First, in Section IV-A, we define collaborative computation which represents the execution of a program in presence of an environment. Subsequently, in Section IV-B, we utilize the collaborative computations to define collaborative stabilization.

**A. Collaborative Computation**

To define collaborative computations, we revisit the requirements of the shepherding program.

- We assume that in each step, either the program (farmer and dog) or the environment (sheep) can make a move.
- If the sheep is given unlimited freedom then it would be impossible to solve the shepherding problem. In other words, it is essential that the farmer and dog make some moves between two moves by the sheep. We generalize this constraint by requiring that the farmer and dog can make at least $k$ moves between two moves by the sheep.

- In the shepherding program, it is permissible for the farmer and dog to reach a situation where they decide to wait. In other words, they do not make any move but wait for the sheep to make the next move. We characterize this constraint by requiring that if the program reaches a state where it has no feasible transition, but the environment has a transition in that state, then the environment must execute some transition in that state.

Hence, we define the computation of a program in presence of an environment as follows:

**Definition 8 (Collaborative Computation):** Let $p$ be a program with state space $S_p$ and transitions $\delta_p$. Let $\delta_e$ be an environment for program $p$ and $k$ be an integer greater than 1. We say that a sequence $\langle s_0, s_1, \ldots \rangle$ is a collaborative computation for program $p$ in presence of environment $\delta_e$ denoted as $\langle p \rangle_{k \delta_e}$-computation if

- $\forall i : i \geq 0 : s_i \in S_p$, and
- $\forall i : i \geq 0 : (s_i, s_{i+1}) \in \delta_p \cup \delta_e$, and
- $\forall i : i \geq 0 : ((s_i, s_{i+1}) \in \delta_e) \Rightarrow (\forall l : l < i < i + k : (\exists s'_l : (s_l, s'_l) \in \delta_p) \Rightarrow (s_i, s_{i+1}) \in \delta_p)$.

$\langle s_0, s_1, \ldots \rangle$ is either infinite, or it ends at $s_k$ such that $\exists s' : s' \in S_p : (s_k, s') \in \delta_p \cup \delta_e$.

From the above definition, it follows that in every step, a program or an environment transition is executed. Moreover, after an environment action, the program has priority for next $k - 1$ steps. In other words, in these steps, the environment executes only if the program has no available transition in the state. In addition, the last constraint requires that a sequence should be a maximal sequence, i.e., it must be either infinite, or ends in a state without any transition in $\delta_p$ or $\delta_e$.

**B. Definition of Collaborative Stabilization**

Before we provide the formal definition, we analyze the requirements of collaborative stabilization in the context of the shepherding example.

A solution to the shepherding problem is an algorithm for the farmer and dog that no matter of the initial location of the farmer, dog, and sheep, always makes the sheep go to location $(1, 4)$. Thus, the set of states where sheep is in location $(1, 4)$ are our desired states. In addition, in the un gated version of the problem, once the sheep is in the location $(1, 4)$ it should not leave the vicinity of location $(1, 4)$. Thus, in this case states where $|s\.row - 1| + |s\.col - 4| \leq 1$ are states that represent the acceptable perturbation from our desired states. We call the set of desired states the invariant, and set of states that the program can reach once it starts from the invariant the safe set.

Following this intuitive example of the shepherding problem, we define a collaborative stabilizing program for an invariant and a safe set as the program that no matter of its initial state, all of its collaborative computations reach a state in the invariant, and once it starts from the invariant it never reaches a state outside the safe set. Figure 2 represents the behavior of a collaborative stabilizing program.

We now formally define collaborative stabilization as follows:

**Definition 9 (Collaborative Stabilization):** Given program $p$ with state space $S_p$, and set of transitions $\delta_e$, state predicates $S$, and $S'$, set of transitions $\delta_e$, and integer $k$ greater than 1, we say program $p$ is $k$-collaborative stabilizing for invariant $S$, safe set $S'$, and environment $\delta_e$ iff following conditions hold:

- $S \subseteq S'$, and
- $\exists (s_0, s_1) : (s_0, s_1) \in \delta_p : s_0 \in S_1 \wedge s_1 \notin S_1$, and
- for any $\langle p \rangle_{k \delta_e}$-computation $\langle s_0, s_1, \ldots \rangle$ there exists $l$ such that $s_l \in S$, and
- for any $\langle p \rangle_{k \delta_e}$-computation $\langle s_0, s_1, \ldots \rangle$ such that $s_0 \in S$, $\forall i : i > 0 : s_i \in S'$.

In the context of the shepherding problem, the rectangle of Figure 2 represents the all $36 \times 36 \times 16$ possible states of the program. The inner circle represents the invariant including our desired states. Thus, all states in the inner circle represent the case where sheep is in location $(1, 4)$. Finally, the outer circle represents the safe set. In the gated version of the problem the outer circle is identical to the inner circle. On the other hand, in the un gated version of the problem, the outer circle is bigger than the inner circle, and all states in it represent the case where sheep is in location $(1, 4)$ or its vicinity.

**V. A COLLABORATIVE STABILIZING PROGRAM FOR THE SHEPHERDING PROBLEM**

In this section, as an example of a collaborative stabilizing program, we provide a program that solves the shepherding problem. For sake of space, we only focus on the gated version of the problem where the safe set is identical to the invariant.

To steer the sheep to the desired location, the farmer and dog should use sheep movement. However, the sheep may not move in the favor of farmer and dog. Figure 3 represents a scenario where the farmer and dog race the sheep forever and
the program never reaches the desired state. To avoid such scenario, the farmer and dog should limit the non-determinism of the sheep movement by exploiting the fact that sheep scares from them. Figure 4 shows how a successful program can prevent the scenario of Figure 3.

Solution. The general strategy of the program is as follows:

- **Preparing:** the farmer first tries to move himself to the left of the sheep. Similarly, the dog tries to move himself to the bottom of the sheep.
- **Adjusting:** The farmer, then, tries to adjust its row with the row of the sheep. Similarly, the dog tries to adjust its column with the column of the sheep.
- **Approaching:** They, then, start to approach the sheep. As they approach the sheep, they force the sheep to go to the desired location.

In each program transition, the farmer and the dog can make up to one move. They select their move based on the moves listed above. For the sake of space, we do not write all program actions here, but as an example we provide the action for the case where the farmer selects its first move and the dog selects its second move from the above lists, i.e., the farmer is in the preparing phase, and the dog is in adjusting phase:

\[
\langle f.col < s.col, s.row > f.row \rangle \rightarrow \langle s.col = 4, d.col := f.col - 1, d.row := f.row + 1 \rangle
\]

Let \( A_p \) represent the set of all program actions obtained as described above. And, let \( \delta_p \) represents the corresponding set of transitions of the environment actions defined in Section III. Then, the program \( p = (V_p, A_p) \) is a 2-collaborative stabilizing for the invariant \( S = (s.row = 1 \land s.col = 4) \), and safe set \( S' = S \), in environment \( \delta_e \). We can design a program for the ungated version where \( S' \neq S \) with the similar approach. However, for that version, we need to use a stronger invariant that requires the farmer and dog to be close to location \((1, 4)\), otherwise it is impossible to guarantee the remaining in the safe set.

In Section VII, we provide a verification method to verify a given program is collaborative stabilizing. Hence, we can verify that the program provided here is actually a collaborative stabilizing program.

VI. RELATION BETWEEN PASSIVE, ACTIVE, AND COLLABORATIVE STABILIZATION

In this section, to understand the difference between collaborative stabilization and other types of stabilization currently proposed in the literature, we compare them with collaborative stabilization. Specifically, we compare collaborative stabilization with conventional stabilization [1] (called passive stabilization in this paper) and active stabilization [2]. In Section VI-A, we provide the definitions of the passive and active stabilization, and in Section VI-B, we provide the comparison.

A. Passive and Active Stabilization

**Passive stabilization.** Passive stabilization is the notion of self-stabilization proposed by Dijkstra [1]. To define passive stabilization, we first define pure-computation as follows:

**Definition 10 (Pure-computation):** Let \( p \) be a program with state space \( S_p \) and transitions \( \delta_p \). We say sequence \( (s_0, s_1, s_2, \ldots) \) is a pure-computation of \( p \) iff

- \( \forall i : i \geq 0 : s_i \in S_p \)
- \( \forall i : i \geq 0 : (s_i, s_{i+1}) \in \delta_p \)
- \( (s_0, s_1, \ldots) \) is either infinite, or it ends at \( s_l \) such that \( \exists s' : s' \in S_p \land (s_l, s') \in \delta_p \).

We define closure as:
Definition 11 (Closure): A state predicate \( S \) is closed in a set of transitions \( \delta \) iff \((\forall (s_0, s_1) : (s_0, s_1) \in \delta : (s_0 \in S \Rightarrow s_1 \in S))\).

Now, we define passive stabilization as follows:

**Definition 12 (Passive Stabilization):** Let \( p \) be a program with state space \( S_p \) and transitions \( \delta_p \). We say program \( p \) is (passive) stabilizing for invariant \( S \) iff

- \( S \) is closed in \( \delta_p \), and
- for any sequence \( \sigma = (s_0, s_1, s_2, \ldots) \) if \( \sigma \) is a pure-computation of \( p \) then there exists \( l \) such that \( s_l \in S \).

*Active stabilization.* Active stabilization is a variation of stabilization for systems where there exists an active adversary that continuously perturbs the program, and disrupts the recovery to the invariant. The notion of active stabilization is proposed in [2]. To facilitate this comparison, we recall the definition of adversary and active stabilization from [2], next.

The adversary is modeled using a set of transitions:

**Definition 13 (Adversary):** We define an adversary for program \( p = \langle S_p, \delta_p \rangle \) to be a subset of \( S_p \times S_p \).

Next, we define a computation of a program in the presence of an adversary:

**Definition 14 (**\( p \)[|k, \text{adv}]**-computation):** Let \( p \) be a program with state space \( S_p \) and transitions \( \delta_p \). Let \( \text{adv} \) be an adversary for program \( p \). And, let \( k \) be an integer greater than 1. We say that a sequence \( (s_0, s_1, s_2, \ldots) \) is a \( \langle p \mid k \rangle \text{adv} \)-computation iff

\[
\begin{align*}
&\forall j \geq 0 :: s_j \in S_p, \text{ and} \\
&\forall j \geq 0 :: (s_j, s_{j+1}) \in \delta_p \cup \text{adv}, \text{ and} \\
&\forall j \geq 0 :: ((s_j, s_{j+1}) \notin \delta_p) \Rightarrow (\forall l : j < l < j + k : (s_l, s_{l+1}) \in \delta_p) \\
&\langle s_0, s_1, \ldots \rangle \text{ is either infinite, or it ends at } s_l \text{ such that } \bar{s}' : s' \in S_p : (s_l, s') \in \delta_p.
\end{align*}
\]

Now, we define active stabilization as follows:

**Definition 15 (Active Stabilization):** Let \( p \) be a program with state space \( S_p \) and transitions \( \delta_p \). Let \( \text{adv} \) be an adversary for program \( p \). And, let \( k \) be an integer greater than 1. We say program \( p \) is \( k \)-active stabilizing with adversary \( \text{adv} \) for invariant \( S \) iff

- \( S \) is closed in \( \delta_p \), and
- \( S \) is closed in \( \text{adv} \), and
- for any sequence \( \sigma = (s_0, s_1, s_2, \ldots) \) if \( \sigma \) is an \( \langle p \mid k \rangle \text{adv} \)-computation then there exists \( l \) such that \( s_l \in S \).

B. Comparison

Revisiting the shepherding program, we argue that active or passive stabilization cannot be utilized to model the requirements of the shepherding program. Specifically, we cannot model the movement of the sheep as program actions. If we model the movement of the sheep as program actions, then in passive stabilization we permit the sheep to make arbitrary number of moves. Also, if we consider the problem of synthesizing a stabilizing protocol for the shepherding program, the sheep needs to be treated differently from the farmer and the dog. In particular, the actions of the farmer and dog can be changed to achieve stabilization, but actions of the sheep cannot be affected by the program.

Regarding the active stabilization, if we attempt to utilize the notion of active stabilization, then the movement of the sheep needs to be modeled as an adversary. However, this is incorrect since the convergence requires the movement of the sheep. By contrast, we cannot require the actions by an adversary in the context of active stabilization.

To summarize, passive stabilization is required when the environment1 is terminating, active stabilization is required when the environment is an enemy, and collaborative stabilization is required when the environment is a frenemy, essential but potentially disruptive. In spite of these differences, there is some relation between passive/active/collaborative stabilization. We identify this relation, next.

Passive stabilization is a special case of collaborative stabilization when the set of environment transitions is empty. Specifically, we have the following observation:

**Observation 1:** If program \( p \) is a passive stabilizing program for invariant \( S \), then \( p \) is \( k \)-collaborative stabilizing for invariant \( S \), safe set \( S' \), in environment \( \delta_e \), where \( \delta_e = \{ \}, k > 1 \), and \( S' = S \).

Active stabilization is also a special case of collaborative stabilization. Specifically, we have the following observation:

**Observation 2:** If program \( p \) is \( k \)-active stabilizing with adversary \( \text{adv} \) for invariant \( S \), then \( p \) is \( k \)-collaborative stabilizing for invariant \( S \), safe set \( S' \), in environment \( \delta_e \), where \( \delta_e = \text{adv} \) and \( S' = S \).

VII. VERIFICATION OF COLLABORATIVE STABILIZING PROGRAMS

In this section, we show that we can verify a collaborative stabilizing program in polynomial time in the state space of that program. The formal statement for the verification problem is as follows:

**Instance.** A program \( p \), with state space \( S_p \), and transitions \( \delta_p \), an environment \( \delta_e \) for \( p \), state predicates \( S \) and \( S' \), and an integer \( k > 1 \).

**Verifying \( k \)-collaborative stabilization decision problem (VkCS).** Is \( p \) \( k \)-collaborative stabilizing for invariant \( S \), safe set \( S' \), in environment \( \delta_e \)?

For program \( p \), environment \( \delta_e \), and integer \( k \), we define a program \( \text{count}_{p, \delta_e, k} \) as follows:

**Definition 16 (**\( \text{count}_{p, \delta_e, k} **):** Let \( p \) be a program with state space \( S_p \) and transitions \( \delta_p \). \( \delta_e \) be an environment for program \( p \), and \( k \) be an integer greater than 1. We define \( \text{count}_{p, \delta_e, k} \) to be the program where:

- **State space:** \( S_{\text{count}_{p, \delta_e, k}} = \{ (s, x) | s \in S_p \land 0 \leq x \leq k - 1 \} \)

- **Transitions:**
  \[
  \delta_{\text{count}_{p, \delta_e, k}} = \{ ((s, s'), (s', x')) | (s, s') \in \delta_p \land x' = \max(x + 1, k - 1) \lor (s, s') \in \delta_e \land x = k - 1 \land x' = 0 \lor (s, s') \in \delta_e \land 0 \leq x < k - 1 \land x' = 0 \land s'' \in S_p : (s, s'') \in \delta_p \}.
  \]

Observe that the state space of \( \text{count}_{p, \delta_e, k} \) consists of tuples where the first element identifies states of \( p \) and the second element captures the number of program transitions that have been executed since the last time that an environment transition executed. Thus, each program transition increases the

---

1Generally, in passive stabilization, a terminating environment is treated as a fault that eventually stops.
second element by one, and each environment transition resets it to 0.

An environment transition can execute only in states where either (1) program has executed sufficient number of steps \((k-1)\), or (2) program has no outgoing transition in that state.

The interesting feature of the count_{p, s, k} is that there is a correspondence between the set of collaborative computations of \(p\) and the set of pure-computations of count_{p, s, k}. Specifically, we have the following theorems:

**Theorem 1:** For any sequence \(\sigma = \langle (s_0, x_0), (s_1, x_1), ..., \rangle\), if \(\sigma\) is a pure-computation of count_{p, s, k} then \((s_0, s_1, \ldots)\) is a \(\langle p\rangle[k\delta_e]\)-computation.

**Theorem 2:** For any sequence \(\sigma = \langle s_0, s_1, \ldots \rangle\), if \(\sigma\) is a \(\langle p\rangle[k\delta_e]\)-computation then there exist 0 \(\leq x_0, x_1, \ldots \leq k-1\) such that \(\langle (s_0, x_0), (s_1, x_1), \ldots \rangle\) is a pure-computation of count_{p, s, k}.

Now, using count_{p, s, k} and Theorems 1 and 2, we show we can verify collaborative stabilization in polynomial time in the size of the state space:

**Theorem 3:** VkCS can be solved in polynomial-time in \(|S_p|\).

**Proof:** We show that we can verify each constraint of Definition 9 in polynomial-time. The first two constraints can be easily verified in polynomial time.

According to Theorems 1 and 2, the third constraint is satisfied if all pure-computations of count_{p, s, k} reaches a state in set \(\{(s, x) | s \in S\}\). To verify that, we first verify that \(\{(s, x) | s \in S\}\) is reachable from all states in the state space. It can be done in polynomial time in the size of the state space. In addition, we need to verify that there is no cycle in the subgraph of count_{p, s, k} excluding states in \(\{(s, x) | s \in S\}\). It can also be done in polynomial time in the size of the state space.

According to Theorems 1 and 2, the last constraint is satisfied if there does not exist a pure-computation of count_{p, s, k} that starts in a state \((s, x)\) where \(s \in S\) and reaches a state \((s', x')\) where \(s' \notin S'\). We can do this reachability analysis in polynomial-time in \(|S_p|\).

**VIII. COMPOSITION OF COLLABORATIVE STABILIZING PROGRAMS**

In this section, we focus on composition of collaborative stabilizing algorithms. In particular, we want to show how we can create more complex collaborative stabilizing programs by composing simpler collaborative stabilizing components. We consider two types of compositions namely selection composition in Section VIII-A and hierarchical composition in Section VIII-C.

Before we focus on composing the collaborative stabilizing programs, first, we extend the definition of collaborative stabilization so that a program provides the desired convergence from a subset of states. This extension is similar to the extension of stabilizing programs to nonmasking fault-tolerant programs considered in [3]. Specifically, Definition 9 requires recovery from every state in the state space. We can also extend Definition 9 to one that only requires recovery in a certain context:

**Definition 17 (Collaborative Stabilization in a Context):** Let \(C\) be a state predicate. Program \(p = (S_p, \delta_p)\) is \(k\)-stabilizing in context \(C\), for invariant \(S\), safe set \(S'\), in environment \(\delta_c\), iff following conditions hold:

- \(S \subseteq S'\) and
- \(#(s_0, s_1) : (s_0, s_1) \in \delta_p : s_0 \in (C \cap S_1) \land s_1 \in (C - S_1)\),
- for any \(\langle p\rangle[k\delta_e]\)-computation \(\langle s_0, s_1, s_2, \ldots \rangle\) where \(\forall i : i \geq 0 : s_i \in C\), there exists \(l\) such that \(s_l \in S\),
- for any \(\langle p\rangle[k\delta_e]\)-computation \(\langle s_0, s_1, s_2, \ldots \rangle\) where \(s_0 \in S \land \forall i : i > 0 : s_i \in C\),

Let \(p_1\) and \(p_2\) be two programs that are collaboratively stabilizing. The property of collaborative stabilization would be proved with respect to the own variables of \(p_1\) and \(p_2\) respectively. When we compose these two programs, we obtain a program, say \(p_3\), whose variables consist of the union of the variables of these two programs. Now, we need to naturally extend the collaborative stabilization property of \(p_1\) to this new composed program (which consists of more variables).

To facilitate such extension, we define the notion of state-transition projection so that we can view the states, transitions, and computations of the composed program through the eyes of \(p_1\) or \(p_2\) respectively.

**Definition 18 (State Projection):** Let \(p = (V_p, A_p)\) and \(p' = (V_{p'}, A_{p'})\) be two programs. The projection of state \(s \in S_p\) on program \(p'\) denoted as \(s[p']\) is \(s' \in S_{p'}\) such that \(\forall v : v \in V_{p'} \land V_p : v(s) = v(s')\).

Observe that in case of the composed programs, the above definition would be instantiated where \(p\) will be the composed program \(p_3\) in the above example, and \(p'\) will be one of the components, say \(p_1\). Also, this definition can be applied in reverse, i.e., from component \(p_1\) to the composed program \(p_3\). In this case, each state \(s\) of \(p_1\) will map to a set of states in \(p_3\) where the values of the variables that are included in \(p_1\) are equal to those specified by state \(s\). However, other variables could be arbitrary.

We can also extend this definition to a set of states (i.e., state predicates) where the projection of a state predicate on program \(p\) is obtained by projecting each state in the state predicate.

We also extend this to sequences (which can be computations in the absence/presence of environments). In this extension, the values of new variables can be arbitrary in the initial state. However, these values will remain unchanged in the subsequent states. Also, by extending this to sequences of length 2, we can use projection on transitions including program/environment transitions. We extend it to the set of transitions/sequences in the same fashion.

**Definition 19 (Sequence Projection):** Let \(p = (V_p, A_p)\) and \(p' = (V_{p'}, A_{p'})\) be two programs. Let \(\sigma = (s_0, s_1, \ldots)\) be a sequence of states of \(p\). The projection of \(\sigma\) over \(p'\) denoted as \(\sigma[p']\) is \((s_0'[p'], s_1'[p'], \ldots)\) such that \(\forall v, i : v \in V_{p'} \land V_p : v(i) = 0 : v(s') = v(s)\) and \(\forall v, i : v \in V_{p'} \land i > 0 : v(s_i') = v(s_i)\).

**A. Selection Composition**

**Definition 20 (Selection Composition):** The selection composition of two programs \(p_1 = (V_{p_1}, A_{p_1})\) and \(p_2 = (V_{p_2}, A_{p_2})\) with selection predicate \(P\) over variables \(V\), is a program denoted by \(p_1 \triangleleft P \triangleright p_2\) with the following sets of variables and actions:

- \(V_{p_1 \triangleleft P \triangleright p_2} = V_{p_1} \cup V \cup V_{p_2}\)
- \(A_{p_1 \triangleleft P \triangleright p_2} = \{((G \land P) \rightarrow C)((G \rightarrow C) \in A_{p_1}) \cup \{((G \land -P) \rightarrow C)((G \rightarrow C) \in A_{p_2})\}\)

If program \(p_1\) is collaborative stabilizing, then for any \(p_2\),
and environment $\text{invariant}$ $(v = 55)$, safe set $(v \in \{54, 55, 56\})$, in environment $\delta_c$ in context $\neg$freeway.

C. Hierarchical Composition

Hierarchical composition considers the scenario where the second program executes only after the first program reaches a state from where it cannot execute further (i.e., the guards of all of its actions are false). Specifically,

**Definition 21 (Hierarchical Composition):** The hierarchical composition of two programs $p_1 = (V_{p_1}, A_{p_1})$ and $p_2 = (V_{p_2}, A_{p_2})$ is a program denoted by $p_1; p_2$ with the following sets of variables and actions:

- $V_{p_1; p_2} = V_{p_1} \cup V_{p_2}$
- $A_{p_1; p_2} = \{(G \to C)((G \to C) \in A_{p_1}) \cup \{(G \land \text{idle}, p_1) \to C((G \to C) \in A_{p_2})\}$

where $\text{idle}, p \equiv \forall(G \to C) : (G \to C) \in A_{p} : \neg G$.

If program $p_1$ is collaborative stabilizing for invariant $S_1$, and it is never idle outside $S_1$. Then, for any program $p_2$ that is closed in $S_1$, $p_1; p_2$ is collaborative stabilizing for invariant $S_1((p_1; p_2))$. Specifically, we have the following theorem (proof is provided in the Appendix):

**Theorem 6:** If

- $p_1$ is a k-collaborative stabilizing program for invariant $S_1$, safe set $S_1$ in environment $\delta_c$, and
- $\forall \delta : S \not\subset S_1 : \text{idle}, p_1$ is not true in $S$, and
- $\delta(s_0, s_1) : (s_0, s_1) \in \delta_2 : s_0 \in S_1 \land s_1 \not\in S_1$, then

$p_1; p_2$ is a k-collaborative stabilizing program for invariant $S_1((p_1; p_2))$, safe set $S_1((p_1; p_2))$ in environment $\delta_c((p_1; p_2))$.

If program $p_1$ is collaborative stabilizing for invariant $S_1$, it is never idle outside $S_1$, and it is idle inside $S_1$. Then, for any program $p_2$ that is collaborative stabilizing for invariant $S_2$, and is closed in $S_1$, $p_1; p_2$ is collaborative stabilizing for invariant $S_1((p_1; p_2)) \cap S_2((p_1; p_2))$. Specifically, we have the following theorem (proof is provided in the Appendix):

**Theorem 7:** Let $p_1$ and $p_2$ be two programs, and $\delta_e$ be an environment for $p_1; p_2$.

- $p_1$ is a k-collaborative stabilizing program for invariant $S_1$, safe set $S_1$ in environment $\delta_e$, and
- $p_2$ is a k-collaborative stabilizing program for invariant $S_2$, safe set $S_2$ in environment $\delta_e$, and
- $\delta(s_0, s_1) : (s_0, s_1) \in \delta_2 : s_0 \in S_1 \land s_1 \not\in S_1$, and
- $\forall \delta : S \not\subset S_1 : \text{idle}, p_1$ is not true in $S$, and
- $\forall \delta : S \subset S_1 : \text{idle}, p_1$ is true in $S$,

then $p_1; p_2$ is a k-collaborative stabilizing program for invariant $S_1((p_1; p_2)) \cap S_2((p_1; p_2))$, safe set $S_1((p_1; p_2)) \cap S_2((p_1; p_2))$ in environment $\delta_e$, and $s_0 \in S_1((p_1; p_2)) \land s_1 \not\in S_1((p_1; p_2))$.

In the above theorem, $\delta_e - \delta_0$ is the environment that inside $S_1$ does not perturb $p_1$, as if the environment perturb $p_1$ from $S_1$, then reaching $S_1((p_1; p_2)) \cap S_2((p_1; p_2))$ cannot be guaranteed.

These theorems can also be extended to cases where the underlying components are collaborative stabilizing only in some context. Using extensions of these theorems in contexts,
we can design and verify hierarchical collaborative stabilizing programs where the program at one level of hierarchy depends upon the stabilization of components in the lower level hierarchy. As an illustration, we can use the extension of these theorems in contexts to further evaluate collaborative stabilization property of a program such as \((p_1; p_2); p_3\) or program such as \(p_3; (p_2 \circ P > p_1)\) and so on.

D. Illustrating Hierarchical Composition

As an example of the hierarchical composition, we consider an adaptive cruise controller. This adaptive cruise controller has two goals: 1) keep the distance with the front car to be within an acceptable value, 2) keep the speed at a desired value. However, the second goal is secondary to the first one.

In other words, if there is a conflict in achieving both goals, the adaptive cruise controller tries to satisfy the first goal. Now, we show how we can create an adaptive cruise controller by composing a simple cruise controller, \(p_e(x)\), and a simple distance controller, \(p_d\), with the following definition:

**Component** \(p_d(x)\). \(p_d(x)\) is a distance controller that changes the speed to keep the distance with the front car equal or greater than \(x\). It is defined as follows:

- \(V_{p_d} = \{v, v_f, d_f\}\)
- \(A_{p_d} = \{(d_f < x \land v \geq v_f) \rightarrow (v = v - 1)\}\)

We also consider the following set of environment actions for the adaptive cruise controller with the set of variables \(\{v, v_f, d_f\}\):

\[
A_e = \{ \\
(d + v_f - v \geq 0) \\
\rightarrow (v \in \{ \\
v + 1, \text{ car is on a downhill.} \\
v - 1, \text{ car is on a uphill.} \\
v, \text{ car is on a flat road.} \\
\}) \\
\land \\
v_f \in \{ \\
v_f + 1, \text{ the front car is accelerating.} \\
v_f - 1, \text{ the front car is decelerating.} \\
v_f, \text{ the front car is stable.} \\
\} \\
\land \\
(d = d + v_f - v), \text{ distance changes.} \\
\}
\]

For sake of simplicity, we assume that cars never crash. Thus, if changing the distance leads to the crash, environment does not change the distance. Therefore, environment only executes if \(d + v_f - v \geq 0\). Let \(\delta_e\) represent the set of transitions of \(A_e\).

The projection of \(A_e\) over \(p_e\) is the same as the environment that we had in Section VIII-B that is \(\{true \rightarrow v := v_e + 1, true \rightarrow v := v_e - 1\}\). The projection of \(A_e\) over \(p_2\) is still \(A_e\), as \(p_2\) has the same set of variables \(\{v, v_f, d_f\}\).

It is straightforward to observer following facts:

- For any \(k > 3\), \(p_d(d_{int})\) is a \(k\)-collaborative stabilizing program for invariant \(S_2 = (d_f \geq d_{int})\), safe set \(S'_2 = (d_f > d_{int})\), in environment \(\delta_e[p_1]\), and
- For any \(k > 2\), \(p_e(v_{int})\) is a \(k\)-collaborative stabilizing program for invariant \(S_1 = (v = v_{int})\), safe set \(S'_1 = (v \in \{v_{int} - 1, v_{int}, v_{int} + 1\})\), in environment \(\delta_e[p_1]\), and
- \(\mathcal{F}(s_0, s_1) : (s_0, s_1) \in \delta_{p_d} : s_0|p_c \in S_1 \land s_1|p_c \notin S_1\)
- \(\forall s : s \notin S_1 : idle.p_d\) is not true in \(s\), and
- \(\forall s : s \in S_1 : idle.p_d\) is true in \(s\).

Thus, we can conclude following results:

- According to Theorem 6, for \(k > 3\), \(p_d(d_{int})\) is a \(k\)-collaborative stabilizing program for invariant \(S_1((p_1; p_2),\) safe set \(S'_1((p_1; p_2),\) in environment \(\delta_e\).
- According to Theorem 7, for any \(k \geq 3\), \(p_d(d_{int})\) is a \(k\)-collaborative stabilizing program for invariant \(S_1((p_1; p_2) \cap S_2((p_1; p_2),\) safe set \(S'_1((p_1; p_2) \cap S'_2((p_1; p_2),\) in environment \(\delta_e - \delta_i\), where \(\delta_i = \{(s_0, s_1)(s_0, s_1) \in \delta_e, s_0 \in S_1((p_1; p_2) \land s_1 \notin S_1((p_1; p_2))\)

We can also create more complex stabilizing controller. For example,

- Michigan speed limit keeper: \(p_{Michigan} = p_c(70) \triangleleft freeway \triangleright (p_e(65) \triangleleft devided\_road \triangleright (p_e(25) \triangleleft residential\_area,p_e(55)))\). It tries to keep the speed at the speed limits of the State of Michigan.
- Michigan adaptive speed limit keeper: \(p_d(d_{int})\). \(p_{Michigan}\). It tries to keep the distance with the front car equal or greater than \(d_{int}\), and at same time, if possible, it ties to keeps the speed at the speed limit of the State of Michigan.

IX. RELATED WORK

The notion of stabilization (a.k.a. self-stabilization) first defined in Dijkstra’s seminal work [1] on stabilizing algorithm for token circulation program. Basically, stabilization requires that the program should reach its legitimate states even if it starts from any arbitrary initial state. We referred to the stabilization defined by Dijkstra [1] as passive stabilization. There are several variations of passive stabilization in the literature.

Fault-containment stabilization (e.g., [4], [5]) refers to that stabilization that requires quick recovery to the invariant if one (or respectively small number of) fault occurs. Byzantine stabilization (e.g., [6], [7]) covers the stabilization in scenarios where a subset of processes is Byzantine. FTSS (e.g., [8]) covers the stabilization scenarios where we have permanent crash faults and other transient faults. Multitolerant stabilization (e.g., [9]) requires the program to mask a class of faults while providing stabilization as well. Active stabilization [2] requires recovery to the invariant even if the program is constantly perturbed by an adversary.

All aforementioned stabilization variations require stabilization along with some addition properties. On the other hand, there are other weaker variations as well. Weak stabilization [10], [11] is one example that requires that from any state there should be a recovery path to the invariant. However, the program may have loops outside the invariant. Other examples of stabilization with weaker set of constraints includes nonmasking fault-tolerance, probabilistic stabilization, and pseudo stabilization. The nonmasking fault-tolerance [12], [13], requires the recovery from only a subset of states. The notion of nonmasking fault-tolerance is similar to the notion of
introduced the notion of safe states that capture the program states that are outside the invariant. To address this issue, in this paper, this is often not satisfied when the environments are perturbing the program and cause them to be in some states that are outside the invariant. To address this issue, we introduced the notion of safe states that capture the program and environment behavior outside the invariant.

X. CONCLUSION

In this paper, we defined the paradigm of collaborative stabilization to capture the requirement of stabilization in the presence of a necessary but potentially disruptive environment. The need for such collaborative stabilization arises in several contexts including cyber-physical systems, distributed systems, and so on. We illustrated some example of such systems including shepherding program, cruise controller, pressure cooker system, furnace system as well as their extensions in the form of persuading an evader to reach a desired destination to facilitate its capture, or cell movement due to chemical and mechanical signals.

We illustrated how collaborative stabilization is related to other forms of stabilization. Specifically, collaborative stabilization differs from traditional (passive) stabilization as well as active stabilization. Passive stabilization assumes that environment/fault actions terminate after some time. As shown earlier, this assumption is not satisfied in above examples. It also differs from active stabilization where environment is treated as an adversary. In these systems, the environment is not essential. Even if the environment chooses not to execute the system is not affected in an adverse manner. As demonstrated earlier this assumption is not satisfied in above examples.

Another key difference between passive stabilization and collaborative stabilization is in terms of inequality between the program transitions and environment transitions. Specifically, for collaborative stabilization, it is required that after the environment executes, the program has a priority for a certain number of subsequent steps. This favoritism towards programs is essential to obtain convergence in the programs identified above. For example, in the shepherding program, allowing the farmer/dog to execute more frequently would still solve the problem. However, allowing sheep to execute more frequently may not. In other words, treating the farmer/dog and the sheep in an identical manner is undesirable. By contrast, in passive stabilization, all processes are generally treated in an identical manner.

One way to summarize the difference between passive/active/collaborative stabilization is as follows: Passive stabilization is required when the environment is terminating, active stabilization is required when the environment is an enemy, and collaborative stabilization is required when the environment is a frenemy, essential but potentially disruptive.

We demonstrated that collaborative stabilization can be verified in polynomial time in the state space of the given program. We also considered two types of compositions, selection composition and hierarchical composition, to compose collaborative stabilizing programs. We showed how these compositions allow us to simplify the design and verification of collaborative stabilizing programs.

Some of the open problems in this area include algorithms for addition of collaborative stabilization to an existing program and to compare the quantitative variation in the time required to verify (passive) stabilization and collaborative stabilization.

REFERENCES

APPENDIX

A. Proof of the Verification Method

Proof of Theorem 1:

Proof: We show this by induction on the number of steps.
The base case is trivial. So, we focus on the inductive case. We show that if:

- \( \langle s_0, s_1, \ldots, s_i \rangle \) is a \( \|k\delta_e \) prefix, and

- \( \langle s_0, x_0, (s_1, x_1), \ldots, (s_i, x_i), (s_{i+1}, x_{i+1}) \rangle \) is prefix of \( \text{count}_{p, \delta, k} \),

then

\( \langle s_0, s_1, \ldots, s_i, s_{i+1} \rangle \) is also a \( \|k\delta_e \) prefix.

We know that if \( \forall s \in (s, s') : (s, s') \in \delta_\text{count}_{p, \delta, k} \) then there are two cases for \( (s_i, s_{i+1}) \):

- \( (s_i, s_{i+1}) \in \delta_p \): In this case clearly \( \langle s_0, s_1, \ldots, s_i, s_{i+1} \rangle \) is a \( \|k\delta_e \) prefix.

- \( (s_i, s_{i+1}) \in \delta_e \): There are two subcases under this case:
  - \( x_i = k - 1 \): By construction of \( \text{count}_{p, \delta, k} \), we know that if a prefix reaches a state with \( x = k - 1 \), then the last \( k - 1 \) transitions are in \( \delta_p \). Thus, \( \langle s_0, s_1, \ldots, s_i, s_{i+1} \rangle \) also is a \( \|k\delta_e \) prefix.
  - \( x_i < k - 1 \): By construction of \( \text{count}_{p, \delta, k} \), we know that there should not exists any program transition stating at state \( s_i \). Thus, the environment transition \( (s_i, s_{i+1}) \) can execute. Therefore, \( \langle s_0, s_1, \ldots, s_i, s_{i+1} \rangle \) also is a \( \|k\delta_e \) prefix.

B. Proofs of Selection Composition

Proof of Theorem 4:

Proof: We need to show that all constraints of Definition 17 are satisfied for program \( p_1 \bowtie p_2 \bowtie p_3 \), invariant \( S_1 \bowtie (p_1 \bowtie \bowtie p_2) \bowtie \bowtie p_3 \), safe set \( S'_1 \bowtie (p_1 \bowtie \bowtie p_2) \bowtie \bowtie p_3 \), and context \( P \):

- \( S_1 \bowtie (p_1 \bowtie \bowtie p_2) \bowtie \bowtie p_3 \) is a prefix of \( \text{count}_{p, \delta, k} \).

We show this by construction of \( \text{count}_{p, \delta, k} \). We have \( (s, s') \in \delta_e \wedge \exists x' : (s', x') \in \delta_p \Rightarrow (s, s') \in \delta_{\text{count}_{p, \delta, k}} \). Thus, in this subcase, \( \langle s_1, (s_{i+1}, x_{i+1}) \rangle \) is prefix of \( \text{count}_{p, \delta, k} \).

The proof of Theorem 5 is similar to the proof of Theorem 4.

C. Proofs of Hierarchical Composition

About the hierarchical composition and computation projection, we have the following lemma. For reason of space proof is provided in the Appendix.

Lemma 1: For every \( (p_1 ; p_2) \| A_e \) computation \( \sigma = (s_0, s_1, \ldots) \) where \( \forall i : i \geq 0 : s_i \notin \text{idle.} p_1, s_i = (s_0, s_1, \ldots) \) is a \( p_1 \| A_e \) computation.
Lemma 2: For every \((p_1;p_2)]_k A_e\) computation \(\sigma = (s_0, s_1, \ldots) \) where \(\forall i : i \geq 0 : s_i \in idle, p_1, \sigma p_2 = (p_2)]_k A_e\) computation.

Proof: Similar to that of Lemma 1.

Proof of Theorem 6:
Proof: We need to show that all constraints of Definition 9 are satisfied for program \(p_1;p_2\), invariant \(S_1|\{p_1;p_2\}\), safe set \(S'_1|\{p_1;p_2\}\), and environment \(\delta_e|\{p_1;p_2\}\).

1. \(S_1|\{p_1;p_2\} \subseteq S'_1|\{p_1;p_2\}\): It is clear from \(S_1 \subseteq S'_1\).
2. \(\#(s_0, s_1) : (s_0, s_1) \in \delta_e|\{p_1;p_2\} : s_0 \in S_1|\{p_1;p_2\} \wedge s_1 \notin S'_1|\{p_1;p_2\}\) (Proof by contradiction) Suppose there is such \((s_0, s_1)\). According to the last constraint of the theorem, \((s_0, s_1)\) is in \(\delta_e|\{p_1;p_2\}\), and \((s_0)p_2|s_0|p_2 \in \delta_e|\{p_1;p_2\}\) that starts in \(S_2\) and leaves \(S_2\) (Contradiction, because \(p_2\) is a \(k\)-collaborative stabilizing program for invariant \(S_2\), safe set \(S'_2\), and environment \(\delta_e|\{p_2\}\)).
3. for any \((p_1;p_2)]_k \delta_e|\{p_1;p_2\}\) - computation \((s_0, s_1, s_2, \ldots)\) there exists \(i\) such that \(s_i \in S_1|\{p_1;p_2\}\). (Proof by contradiction) Suppose there exist all constraints of Definition 9 are satisfied for program \(p_1;p_2\), invariant \(S_1|\{p_1;p_2\}\), safe set \(S'_1|\{p_1;p_2\}\), and environment \(\delta_e|\{p_1;p_2\}\).

Proof of Theorem 7:
Proof: We need to show all constraints of Definition 9 are satisfied for program \(p_1;p_2\), invariant \(S_1|\{p_1;p_2\}\), safe set \(S'_1|\{p_1;p_2\} \cap S'_2|\{p_1;p_2\}\), and environment \(\delta_e - \delta_l\), where \(\delta_l = \{(s_0, s_1) : s_0 \in S_1|\{p_1;p_2\} \wedge s_1 \notin S'_1|\{p_1;p_2\}\}\).

1. \(S_1|\{p_1;p_2\} \cap S'_2|\{p_1;p_2\} \subseteq S'_1|\{p_1;p_2\} \cap S'_2|\{p_1;p_2\}\): It is clear from \(S_1 \subseteq S'_1\) and \(S_2 \subseteq S'_2\).
2. \(\#(s_0, s_1) : (s_0, s_1) \in \delta_e|\{p_1;p_2\} : (s_0 \in S_1|\{p_1;p_2\} \cap S'_2|\{p_1;p_2\}) \wedge s_1 \notin S'_1|\{p_1;p_2\} \cap S'_2|\{p_1;p_2\}\) (Proof by contradiction) Suppose there is such \((s_0, s_1)\). According to the last constraint of the theorem, \(p_1\) is idle in the \(S_1\). Thus, \((s_0, s_1)\) is in \(\delta_e|\{p_1;p_2\}\), and \((s_0)p_2|s_0|p_2 \in \delta_e|\{p_1;p_2\}\) that starts in \(S_2\) and leaves \(S_2\) (Contradiction, because \(p_2\) is a \(k\)-collaborative stabilizing program for invariant \(S_2\), safe set \(S'_2\), and environment \(\delta_e|\{p_2\}\)).