ADAPTIVE PROGRAMMING

Mohamed G. Gouda and Ted Herman

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-89-29 October 1989

Abstract

An adaptive program is one that changes its behavior based on the current state of its environment. The environment state is assumed to go through successive periods of change and stability, and the adaptive program is only required to perform its intended function during periods of stability. In this paper, this notion of adaptivity is formalized and a logic for reasoning about adaptive programs is presented. The logic includes several composition operators that can be used to define an adaptive program in terms of given constituent programs; programs resulting from these compositions retain all the adaptive properties of their constituent programs.

1. Introduction

An adaptive program is one that changes its behavior according to its environment: for each environment state one behavior is most appropriate. One way to think about adaptivity is to view the different behaviors of an adaptive program as a belonging to different programs. An *adaptive* program may behave like one program S in one environment state and like another program T in another environment state. For instance, S can be a sequential program that uses only one processor, whereas T can be a parallel program that uses two or more processors. In this case, the adaptive program behaves like S in any environment state where only one processor is available, and like T in any state where the number of available processors is two or more. In an environment where the number of available processors changes over time, the adaptive program changes its behavior between S and T depending on the current number of available processors.

Viewing the different behaviors of an adaptive program as belonging to different programs can be exploited in a modular methodology for program composition. An adaptive program that behaves like S in one environment state and like T in another environment state can in many cases be developed in two steps. First, the two programs S and T are developed as separate independent programs, possibly in parallel. Second, the developed programs are later combined to form the required adaptive program.

Our aim in this paper is to formalize this notion of program adaptivity and to present a formal basis for composing adaptive programs.

Central to our presentation is the notion of an "environment." Informally, the environment of a program is defined by a set of input variables that can be read, but never written by the program. The values of these variables may change arbitrarily over time by an outside agent. We assume, nevertheless, that periods of change are relatively short and are usually followed by long periods of stability where the values of the input variables are constant. It is during these long periods of stability that the program is required to perform its useful function as dictated by the current demands of the environment. There are no requirements on the program's behavior during periods of change.

The relationship between a program and its environment has been investigated by many researchers before. See for instance, Hoare's CSP [5], Milner's CCS [9], Lamport's modules [7], Pnueli's reactive systems [10], Chandy and Misra's conditional properties [2] , Lam and Shankar's interfaces [6], Lynch's I/O automata [8], etc. Most of these investigations are based on the assumption that the environment's behavior, especially its interaction with the program, can be fully predicted in advance, and in some instances fully planned. Under this assumption, the program can be designed to always perform its intended function. By contrast, our work is based on the assumption that the environment's behavior cannot be fully predicted. In particular, the input variables of the program may assume changing values for some time provided that they eventually stabilize into fixed values. In this case, it is not reasonable to expect that the program will always perform its intended function. The best that one can hope for is that the program performs its function only after the values of all input variables have stabilized. It is this premise and its implications that distinguish our study from previous investigations.

The rest of this paper is organized as follows. In Section 2, we define adaptive sequential programs and present rules for reasoning about the adaptive properties of such programs; rules for composing adaptive sequential programs are presented in Section 3. Next, we extend the discussion to concurrent programs (Section 4), and introduce rules for composing adaptive concurrent programs (Section 5). In Section 6, we exercise the composition rules presented earlier to develop an adaptive token program, and in Section 7, we discuss the relationship between adaptivity and self-stabilization. Concluding remarks are in Section 8. Proofs for most of the rules appear in the appendix in Section 9.

2. Adaptivity of Sequential Programs

Let S be a sequential program defined by $S = (V, A)$, where V is a set of *variables* and A is a set of *actions*. Every variable in V has a prescribed domain and is either an *input* variable or an *internal* variable. Each action in A is a guarded assignment of the form $P \rightarrow C$, where P is a predicate over the variables in V, and C is an assignment statement that assigns values to internal variables only.

The *state-space* of S is the cartesian product of the domains of all variables in V. A *state* of S is an element of the state-space; it thus denotes a value for each variable in V. A *state predicate* is a boolean function of the state-space. If the value of a state predicate P is *true* at some state r, then we say P holds at r.

A state predicate P is an *input* predicate in program S, denoted **input** P **in** S (or simply **input** P when S is understood) iff P is a predicate whose definition makes no reference to any internal variable of S.

A *transition* of S is an ordered pair of states (r,s) such that S has an action $P \rightarrow C$, where P holds at r, and s is obtained from r by replacing the values of internal variables as indicated by the assignment statement C. If (r,s) is a transition and P holds at r, then we say $P \rightarrow C$ is enabled at r.

A *computation* is a sequence of states such that every consecutive pair is a transition. Empty and single-state sequences are therefore computations and any prefix or suffix of a computation is a computation. We restrict the scope of prefix and suffix as follows. Any prefix of a computation is a finite sequence; any suffix of a computation is a suffix with respect to a (finite) prefix; any suffix of a non-empty computation is a non-empty computation.

A computation is *maximal* iff it is not a proper prefix of any computation. That is, a maximal computation is either infinite or there exists no transition originating at the last state of the computation.

The central characterization for adaptivity is the following relation "secures" between any two state predicates.

> P **secures** Q **in** S iff
>
> P is an input predicate in S, and for each maximal computation of S: if P holds at each state in the computation then there exists a suffix of the computation such that Q holds at each state in the suffix.

(adaptivity)

For convenience, we write P **secures** Q when S is understood.

Operationally, we interpret P **secures** Q as follows. It is always possible for the environment to lead the program to a state satisfying P by some setting of input variables. Once P is satisfied the program will converge on its own accord to a situation in which Q holds and continues to hold indefinitely, unless the environment forces the program outside Q by some resetting of the input variables.

Example 1. Consider the following program that has three input variables b, c, and d, one internal variable z, and two actions.

input variable b,c,d: **integer**;

internal variable z: **integer**;

actions $(b = 0) \rightarrow z := \max(c,d)$, $(b = 1) \rightarrow z := \min(c,d)$.

For this program we prove $(b = 0)$ **secures** $(z = \max(c,d))$. There are two proof obligations from the definition of **secures** . First, $(b = 0)$ is an input predicate because b is an input variable. Second, consider any maximal computation where $(b = 0)$ holds at each state. Only the first action is enabled at each state of this computation. Thus, the first transition in the computation establishes $(b = 0) \wedge (z = \max(c,d))$ and subsequent transitions leave the state unchanged. This completes our proof of $(b = 0)$ **secures** $(z = \max(c,d))$.

Similarly it can be shown that $(b = 1)$ **secures** $(z = \min(c,d))$. ∎

The following properties of **secures** are stated as inference rules. (Proofs of these properties appear in the Appendix.)

$$\frac{\textbf{input } P}{P \textbf{ secures } \mathit{true}} \qquad \text{(truth)}$$

$$\frac{P \textbf{ secures } Q}{P \textbf{ secures } (P \wedge Q)} \qquad \text{(sharpening)}$$

$$\frac{P \textbf{ secures } Q, \quad \textbf{input } R, \quad R \Rightarrow P}{R \textbf{ secures } Q} \qquad \text{(strengthening)}$$

$$\frac{P \textbf{ secures } Q, \quad Q \Rightarrow R}{P \textbf{ secures } R} \qquad \text{(weakening)}$$

$$\frac{P \textbf{ secures } Q, \quad R \textbf{ secures } T}{(P \vee R) \textbf{ secures } (Q \vee T), \quad (P \wedge R) \textbf{ secures } (Q \wedge T)} \qquad \text{(junction)}$$

Other properties of **secures** can be derived by various combinations of these rules. For instance,

$$\frac{\textbf{input } P}{P \textbf{ secures } P} \qquad \text{(stability)}$$

can be derived from the truth and sharpening rules. Similary,

$$\mathit{false} \textbf{ secures } R \qquad \text{(falsehood)}$$

can be derived from the stability and weakening rules.

3. Composition of Sequential Programs

Large adaptive programs can be composed by combining smaller adaptive programs. This section describes two types of composition: level and hierarchical. *Level composition* combines two adaptive programs so that they have equal roles in the resulting composite program. *Hierarchical composition* combines two adaptive programs so that one of the programs is subordinate to the other. These compositions can be applied repeatedly to combine any finite number of adaptive programs into a single program.

3.1 *Level Composition*

A prerequisite for level composition of two programs is their compatibility. Given two sequential programs S and T, S **compatible** T holds iff all variables with the same name in both S and T are of the same type, and they are either input or internal in both programs. Notice that compatibility of two programs is not difficult to achieve: by renaming variables in one program so that no name is common to both programs, compatibility is assured.

The following definition introduces notation to describe the level composition of two compatible programs. Let $S = (V, A)$ and $T = (W, B)$ be two compatible sequential programs and let e be a fresh boolean input variable that does not occur in S or T. Let $S[e]T$ denote the program (U, E) where

$$U = V \cup W \cup \{e\}, \quad \text{and}$$

$$E = \{(e \wedge C) \rightarrow D \mid C \rightarrow D \in A\} \cup \{(\neg e \wedge C) \rightarrow D \mid C \rightarrow D \in B\}.$$

Based on this definition, the following rule can be proved.

$$\frac{P \textbf{ secures } Q \textbf{ in } S, \quad P \textbf{ secures } R \textbf{ in } T, \quad S \textbf{ compatible } T}{\begin{array}{l}(P \wedge e) \textbf{ secures } Q \textbf{ in } S[e]T, \\ (P \wedge \neg e) \textbf{ secures } R \textbf{ in } S[e]T\end{array}}$$

(level composition)

Example 2. It is required to design a traffic control program. In the "morning" the program directs traffic either right or across, and in the "evening" it directs traffic either left or across. Consequently we propose a program that has an internal variable *traffic* that can take any of the values: left, right, or across. Inspired by our level composition rule, we design two adaptive programs, a morning program S and an evening program T. Each of these two programs contains assignments to the internal variable *traffic*. By combining the two programs using an input variable named *morning*, we construct the required traffic control program as $S[morning]T$.

The morning program S is defined as follows.

input variable *waiting*: **boolean**;

internal variable *traffic*: (left,right,across);

actions $waiting \rightarrow traffic := \text{across}, \quad \neg waiting \rightarrow traffic := \text{right}.$

It can be shown that

$waiting$ **secures** $(traffic = \text{across})$ **in** S, and $\neg waiting$ **secures** $(traffic = \text{right})$ **in** S.

The result can be reduced by the junction and sharpening rules to

$true$ **secures** $((waiting \wedge traffic = \text{across}) \vee (\neg waiting \wedge traffic = \text{right}))$ **in** S.

The evening program T is defined as:

input variable *waiting*: **boolean**;

internal variable *traffic*: (left,right,across);

actions $waiting \rightarrow traffic := \text{across}, \quad \neg waiting \rightarrow traffic := \text{left}.$

By symmetry with S the adaptivity property of T is

$true$ **secures** $((waiting \wedge traffic = \text{across}) \vee (\neg waiting \wedge traffic = \text{left}))$ **in** T.

The required traffic control program $S \,[\![morning]\!]\, T$ is:

input variable *waiting*, *morning*: **boolean**;

internal variable *traffic*: (left,right,across);

actions

$morning \wedge waiting \rightarrow traffic := \text{across},$

$morning \wedge \neg waiting \rightarrow traffic := \text{right},$

$\neg morning \wedge waiting \rightarrow traffic := \text{across},$

$\neg morning \wedge \neg waiting \rightarrow traffic := \text{left}.$

The adaptivity property of program $S \,[\![morning]\!]\, T$ follows from the level composition rule. After simplification the result is

$true$ **secures**

$(\ (waiting \wedge traffic = \text{across})$

$\vee\ (\neg waiting \wedge morning \wedge traffic = \text{right})$

$\vee\ (\neg waiting \wedge \neg morning \wedge traffic = \text{left})\ ).$ ∎

3.2 *Hierarchical Composition*

A prerequisite for hierarchical composition of two programs is the controllability of one of them by the other. For sequential programs S and T, the relation T **controls** S holds iff all variables with the same name in both S and T are of the same type, and no internal variable of S is a variable in T. There is no restriction on the input variables in S: an input variable in S may be an internal or input variable in T.

The following two definitions introduce notation to state the hierarchical composition rule. Define **idle**.S, for a sequential program S, to be the conjunction of the negated guards of all actions of S, i.e.,

$$\textbf{idle}.S \quad \equiv \quad (\forall \text{ action } P \rightarrow D \text{ of } S : \ \neg P)$$

Thus, **idle**.S is satisfied by any state where no action of S is enabled.

Let $S = (V, A)$ and $T = (W, B)$ be sequential programs where T **controls** S. Let $T;S$ denote the program (U, C) where

$$U = (V - W) \ \cup \ W \qquad \text{and}$$

$$C = \{ (\textbf{idle}.T \wedge P) \rightarrow D \mid \ P \rightarrow D \in A \} \ \cup \ B.$$

Based on this definition, the following rule can be proved.

$$\frac{P \textbf{ secures } Q \textbf{ in } T, \quad (P \wedge Q) \Rightarrow \textbf{idle}.T, \quad Q \textbf{ secures } R \textbf{ in } S, \quad T \textbf{ controls } S}{P \textbf{ secures } R \textbf{ in } T;S}$$

(hierarchical composition)

Example 3. Let program U be defined as follows.

input variable *clock*: **integer**;

internal variable *morning*: **boolean**;

actions

$clock > 0 \ \wedge \ clock < 1200 \ \wedge \ \neg morning \ \rightarrow \ morning := true,$

$(clock \leq 0 \ \vee \ clock \geq 1200) \ \wedge \ morning \ \rightarrow \ morning := false.$

It is straightforward to show that program U is suitable for hierarchical composition with the composite program $S[morning]T$ of Example 2. The proof consists of four steps.

First, it can be shown that ($clock > 0 \ \wedge \ clock < 1200$) **secures** *morning* **in** U.

Second, it can be shown that ($clock > 0 \ \wedge \ clock < 1200 \ \wedge \ morning$) $\Rightarrow$ **idle**.U.

The third condition follows by the level composition rule:

morning **secures** ((*waiting* $\wedge$ *traffic* = across) $\vee$ ($\neg$ *waiting* $\wedge$ *traffic* = right))

in $S[morning]T$.

Fourth, the internal variable *traffic* of $S[morning]T$ does not appear in U, therefore U **controls** ($S[morning]T$).

Thus, the hierachical composition rule can be applied to combine the two programs U and $S[morning]T$ into one program $U;(S[morning]T)$ with the result

($clock > 0 \ \wedge \ clock < 1200$)

secures ((*waiting* $\wedge$ *traffic* = across) $\vee$ ($\neg$ *waiting* $\wedge$ *traffic* = right))

in $U;(S[morning]T)$. ∎

4. Adaptivity of Concurrent Programs

We define a concurrent program as a sequential program that meets some additional constraints imposed by partitioning its variable set and its action set. Thereby, the definitions of adaptavity, compatibility and composition in sections 2 and 3 have straightforward extensions for concurrent programs.

A *concurrent program* is a pair (S, α), where $S = (V, A)$ is a sequential program and α satisfies the following two conditions. First, α partitions V into disjoint sets V_i for $0 \le i < \text{rank}(\alpha)$. Second, the actions of A can be partitioned into sets A_i for $0 \le i < \text{rank}(\alpha)$ so that for every $P \rightarrow C$ in A_i the assignment statement C assigns values to variables of V_i only.

In this definition, a concurrent program may be interpreted as a collection of processes. A *process* of a concurrent program (S, α) is a sequential program $S_i = (V_i \cup W_i, A_i)$, where V_i and A_i are

corresponding sets of variables and actions defined by the partition α, such that $0 \le i < \text{rank}(\alpha)$, and W_i is the set of all variables appearing in A_i but not in V_i.

We extend the definition of **secures** to concurrent programs as follows.

P **secures** Q **in** (S, α) iff

(S, α) is a concurrent program and P **secures** Q **in** S.

(concurrency)

From this definition, the above rules for reasoning about the **secures** properties of sequential programs can be extended in a straightforward way for reasoning about the **secures** properties of concurrent programs.

5. Composition of Concurrent Programs

Two concurrent programs are composed by composing their corresponding processes. To this end we extend the definitions of level and hierarchical composition. We also define a new form of level composition called distributed composition.

5.1 Level Composition

(S, α) **compatible** (T, β) holds iff $\text{rank}(\alpha) = \text{rank}(\beta)$, S **compatible** T, and for every variable x assigned by actions of both S and T: x is assigned by an action of S_i iff x is assigned by an action of T_i, for each i, $0 \le i < \text{rank}(\alpha)$.

The level composition of compatible concurrent programs is written $(S, \alpha)\,[e]\,(T, \beta)$, which denotes the concurrent program whose processes are $\{ S_i\,[e]\,T_i \mid 0 \le i < \text{rank}(\alpha) \}$. The rule for level composition of two concurrent programs (S, α) and (T, β) is as follows.

$$\frac{P \text{ secures } Q \text{ in } (S, \alpha), \quad P \text{ secures } R \text{ in } (T, \beta), \quad (S, \alpha) \text{ compatible } (T, \beta)}{\begin{array}{l}(P \wedge e) \text{ secures } Q \text{ in } (S, \alpha)\,[e]\,(T, \beta), \\ (P \wedge \neg e) \text{ secures } R \text{ in } (S, \alpha)\,[e]\,(T, \beta)\end{array}}$$

It is straightforward to show that this rule can be derived from the level composition rule of section 3 and and the definition of concurrency in section 4.

5.2 Hierarchical Composition

(T, β) **controls** (S, α) holds iff $\text{rank}(\alpha) = \text{rank}(\beta)$, T **controls** S, and for every variable x assigned by actions of both S and T: x is assigned by an action of S_i iff x is assigned by an action of T_i, for each i, $0 \le i < \text{rank}(\alpha)$.

The hierarchical composition of concurrent programs is written $(T, \beta) ; (S, \alpha)$, which denotes the concurrent program whose processes are $\{ T_i ; S_i \mid 0 \le i < \text{rank}(\alpha) \}$. The rule for hierarchical composition of two concurrent programs (S, α) and (T, β) is as follows.

$$\frac{P \textbf{ secures } Q \textbf{ in } (T, \beta), \quad (P \wedge Q) \Rightarrow \textbf{idle}.T, \quad (P \wedge Q) \textbf{ secures } R \textbf{ in } (S, \alpha), \quad (T, \beta) \textbf{ controls } (S, \alpha)}{P \textbf{ secures } R \textbf{ in } (T, \beta) ; (S, \alpha)}$$

It is straightforward to show that this rule can be derived from the hierarchical composition rule of section 3 and and the definition of concurrency in section 4.

5.3 Distributed Composition

The level composition of concurrent programs requires that all processes in the composite program read the same input variable to make their individual adaptive choices. This is unreasonable in a distributed implementation. The following definitions introduce notation for a distributed composition rule.

Let $\tilde{e}$ be a vector of boolean input variables, that is, $\tilde{e} = \{ e_i \mid 0 \le i < \text{rank}(\alpha) \}$. We use angular brackets to abbreviate universal quantification over all the elements of $\tilde{e}$, i.e.

$$\langle \tilde{e} \rangle \equiv (\forall i: 0 \le i < \text{rank}(\alpha): e_i);$$

$$\langle \neg \tilde{e} \rangle \equiv (\forall i: 0 \le i < \text{rank}(\alpha): \neg e_i).$$

The level composition of compatible distributed programs is written

$$(S, \alpha) [\![\tilde{e}]\!] (T, \beta),$$

which denotes the concurrent program whose processes are

$$\{ S_i [\![e_i]\!] T_i \mid 0 \le i < \mathrm{rank}(\alpha) \}.$$

The distributed composition rule for two concurrent programs (S, α) and (T, β) is as follows.

$$\frac{P \textbf{ secures } Q \textbf{ in } (S,\alpha), \quad P \textbf{ secures } R \textbf{ in } (T,\beta), \quad (S,\alpha) \textbf{ compatible } (T,\beta)}{\begin{array}{l}(P \wedge <\widetilde{e}>) \textbf{ secures } Q \textbf{ in } (S,\alpha)[\![\widetilde{e}]\!](T,\beta), \\ (P \wedge <\neg\widetilde{e}>) \textbf{ secures } R \textbf{ in } (S,\alpha)[\![\widetilde{e}]\!](T,\beta)\end{array}}$$

(distributed composition)

An example of applying the distributed composition rule is discussed next.

6. Case Study: The Adaptive Token

In this section, we construct an adaptive distributed program for circulating a token in a ring of processes. Two reasonable methods for circulating the token are "busy," in which the token is circulated continuously, and "lazy," in which the token is circulated when and only when at least one process needs the token. Each of these two methods is realized by a separate program; then an adaptive token program is constructed by the distributed composition of the busy and lazy token programs.

6.1 Busy Token

The busy token program, henceforth called *Busy*, continuously circulates one token among a set of n processes:

$$Busy \equiv \{ S_i \mid 0 \le i < n \},$$

where each process S_i is defined as follows.

internal variable x_i : **integer;**

actions $x_i \bmod n = i \;\rightarrow\; x_i := x_i + 1, \qquad x_{i-1} > x_i \;\rightarrow\; x_i := x_{i-1}.$

In the above program and for the remainder of the case study, we adopt the following notational convention. Let x_k, for an arbitrary integer k, refer to variable x_i satisfying $i = (k \bmod n)$. Thus, the second action for process S_0 can be rewritten as

$x_{n-1} > x_0 \quad \rightarrow \quad x_0 := x_{n-1}$.

We say that process S_i holds a token when $x_i \bmod n = i$. The appendix of this paper outlines a proof of the following property for the busy token program.

true **secures** *homebusy* **in** *Busy*

where *homebusy* is a predicate that holds at any state where there is exactly one process S_i satisfying $(x_i \bmod n = i) \ \vee \ ((x_{i-1} \bmod n = i) \ \wedge \ (x_{i-1} > x_i))$. Informally, *homebusy* describes a state where exactly one process either has a token or will have a token immediately after its next transition.

6.2 *Lazy Token*

The lazy token program, henceforth called *Lazy*, circulates one token among a set of n processes when and only when one or more of the processes needs a token.

$Lazy \ \equiv \ \{ T_i \mid 0 \le i < n \}$,

where each process T_i is defined as follows.

internal variable x_i : **integer;** y_i : **boolean;**

input variable z_i : **boolean;**

actions

$x_i \bmod n = i \ \wedge \ y_{i+1} \quad \rightarrow \ (x_i, y_i) := (x_i + 1, \mathit{false})$,

$x_{i-1} > x_i \quad \rightarrow \ x_i := x_{i-1}$,

$\neg y_i \ \wedge \ y_{i+1} \ \wedge \ \neg m_i \quad \rightarrow \ y_i := \mathit{true}$,

$\neg y_i \ \wedge \ z_i \quad \rightarrow \ y_i := \mathit{true}$.

Each process T_i has an integer variable x_i and two boolean variables y_i and z_i. Variable z_i is an input variable indicating the need for a token by its process T_i. We say that process T_i holds a token when $x_i \bmod n = i$. The following predicate is used in the guard of one action:

$m_i \ \equiv \ (x_i \bmod n = i) \ \vee \ (x_{i+1} \bmod n = (i+1)) \ \vee \ (x_i \bmod n = (i+1) \ \wedge \ x_i > x_{i+1})$.

The predicate m_i holds when T_i has a token, or T_{i+1} has a token, or when there is a token between T_i and T_{i+1}.

The adaptivity property for this program is

true **secures** *homelazy* **in** *Lazy*

where the *homelazy* predicate holds at any state where there is exactly one process T_i satisfying $(x_i \bmod n = i) \ \vee \ ((x_{i-1} \bmod n = i) \ \wedge \ (x_{i-1} > x_i))$.

6.3 *Adaptive Token*

The busy and lazy token programs are compatible; hence they can be composed using the distributed composition rule. The resulting composite program $Busy \ [\tilde{e}] \ Lazy$ is then guaranteed to satisfy the following properties:

$<\tilde{e}>$ **secures** *homebusy*, and

$<\neg \tilde{e}>$ **secures** *homelazy*.

Notice that the two variables, x_i in process S_i, and x_i in process T_i, are replaced by only one variable x_i in the composite process $S_i \ [e_i] \ T_i$. Thus, the distributed composition rule tends to reduce the number of variables in the resulting composite programs.

7. Adaptivity and Self-Stabilization

A self-stabilizing program [1, 3] is a particular type of adaptive program; but before we formally state the relationship between adaptivity and self-stabilization, we need to define self-stabilization in our model of computation.

For a sequential program S and predicate Q, S **self-stabilizes to** Q iff each maximal computation of S can be partitioned into a prefix and a suffix where each state in the prefix satisfies $\neg Q$ and each state in the suffix satisifes Q.

The relationship between self-stabilization and adaptivity is stated by the rule

$$\frac{S \ \textbf{self-stabilizes to} \ Q}{\textit{true} \ \textbf{secures} \ Q \ \textbf{in} \ S} \qquad \text{(self-stabilization)}$$

which can be easily proven from the defintions of "secures" and "self-stabilization."

The converse of this rule does not hold in general. That is, *true* **secures** Q in a program does not imply that the program self-stabilizes to Q. For example, program *Busy* in section 6.1 satisfies

true **secures** *homebusy*, whereas it is not the case that *Busy* **self-stabilizes to** *homebusy* for the following reason. Program *Busy* has a maximal computation where there is only one token in the initial state, but after some transitions more tokens appear. Thus, although the computation does have an infinite suffix of single-token states, it does not satisfy the definition of self-stabilization.

It is tempting to conjecture that if *true* **secures** Q holds for a program S then S **self-stabilizes to** R for some predicate R where $R \Rightarrow Q$. The busy token program does satisfy this conjecture. The rationale for the conjecture is the following. From *true* **secures** Q we assert that each maximal computation has a suffix where Q holds at each state in the suffix. Consequently, one could hope to construct some predicate R to characterize only those states in the suffix. But the conjecture is falsified by the following example.

Let Z be the program

internal variable x : **domain** (0, 1, 2);

actions $x = 0 \rightarrow x := 0,$

$x = 1 \rightarrow x := 1,$

$x = 1 \rightarrow x := 2,$

$x = 2 \rightarrow x := 0.$

It is straightforward to show that *true* **secures** $(x \neq 2)$ **in** Z . Any maximal computation whose initial state is $x = 0$ or $x = 2$ has a suffix where $x = 0$ at every state. Any maximal computation whose initial state is $x = 1$ has a suffix where either $x = 1$ at every state or $x = 0$ at every state.

We now refute the possibility of some R satisfying Z **self-stabilizes to** R and $R \Rightarrow (x \neq 2)$. Expansion of $R \Rightarrow (x \neq 2)$ yields four possibilities for R: *false*, $(x = 0)$, $(x = 1)$, and $(x \neq 2)$. In each case we exhibit a maximal computation that cannot be partitioned to satisfy Z **self-stabilizes to** R. First, Z does not self-stabilize to *false* because maximal compuations of Z are non-empty. Second, Z does not self-stabilize to $(x = 0)$ because there is a maximal computation consisting of $(x = 1)$ for all its states. Third, Z does not self-stabilize to $(x = 1)$ because there is a maximal computation consisting of $(x = 0)$ for all its states. Fourth, Z does not self-stabilize to $(x \neq 2)$ due to following maximal computation: a non-empty sequence of $(x = 1)$ states, followed by a $(x = 2)$ state, followed by an infinite sequence of $(x = 0)$ states. (end of refutation).

8. Conclusions

The main contribution of this paper is to identify **secures** as the basic concept for program adaptivity, and to present a simple, yet effective, logic for reasoning about the **secures** properties of adaptive programs. The highlights of our logic are three rules for composing large adaptive programs by combining given constituent programs. The first two rules, level and hierarchical compositions, apply to both sequential and concurrent programs, while the last rule, distributed composition, applies only to concurrent programs.

So far we have assumed that the values of input variables remain fixed after the environment has stabilized. Extending the work to accomodate the important possibility that the values of these variables may change, according to a given protocol, during periods of stability has been investigated in [4].

Acknowlegements: We would like to thank Chris Lengauer, Jayadev Misra, Simon Lam, Louis Rosier, and Ambuj Singh for reading earlier drafts of this manuscript and providing valuable advice.

9. Appendix

9.1 Proofs of Inference Rules

Each of the proofs in this section is based on the definition of adaptivity, that is P **secures** Q for some form of P and Q. From the definition of **secures** there are two proof obligations. The first obligation is to establish that P is an input predicate; this is a trivial task of verification that we omit. The second obligation is to show that every maximal computation has an appropriate suffix where Q holds at each state -- this is the part of the proof we present in each case below. To further streamline the presentation, we observe that there are two cases for a maximal computation, either it is empty or non-empty. In case it is empty the definition reduces to universal quantification over an empty range and **secures** holds trivially. Therefore maximal computations are assumed to be non-empty in the proofs.

Truth: Consider any maximal computation such that P holds at each state. Observe that $P \Rightarrow true$ is a tautology, so *true* holds at all states.

∎

Sharpening: From the antecedent, in each maximal computation where P holds at each state there exists a suffix in which Q holds at each state; therefore $(P \wedge Q)$ holds at each state in the suffix.

∎

Strengthening: Consider a maximal computation τ in which R holds at each state. From $R \Rightarrow P$ it follows that P also holds at each state in τ. The antecedent P **secures** Q implies that Q holds in at each state in some suffix of τ. ∎

Junction (conjunction): Consider a maximal computation τ in which $(P \wedge R)$ holds at each state. Observe that $(P \wedge R) \Rightarrow P$ and $(P \wedge R) \Rightarrow R$, so both antecedents are applicable; τ has a suffix where Q holds at each state and τ has a suffix where T holds at each state, therefore τ has a suffix where $(Q \wedge T)$ holds at each state.

∎

Junction (disjunction): Consider a maximal computation τ in which $(P \vee R)$ holds at each state. Let (r,s) be some transition in τ. There are three cases for r, either (i) $P \wedge \neg R$ holds at r, (ii) $\neg P \wedge R$ holds at r, or (iii) $P \wedge R$ holds at r. The predicates P and R are input predicates, so the case (i-iii) for state s is the same as the case for r -- input variable values do not change due to transitions. Moreover, one case (i-iii) holds for all states in τ. For case (i), P holds at each state in τ, and the antecendent asserts there is a suffix where Q holds at each state, hence $Q \vee T$ holds at each state of the suffix. The treatment of case (ii) follows by symmetry. Case (iii) follows from conjunction, shown above, and the fact $(Q \wedge T) \Rightarrow (Q \vee T)$. ∎

9.2 Proofs of Composition Rules

For proofs of the composition rules we define state projections. A projection maps a state of a composite program to a state of one of its constituent programs. For instance, let $S \circ T$ be some

composition of the two programs S and T; let h be the projection from a state of $S \circ T$ to a state of S: h excludes variables that appear in $S \circ T$ but not in S. We extend projections to operate on sequences of states by element-wise application. Observe that if ρ is a sequence of states of $S \circ T$ and P is a predicate over the variables of S, then P holds at each state in ρ iff P holds at each state in $h(\rho)$.

Level Composition: The two parts of the rule's conclusion are symmetric, so we demonstrate one part only. Let f be the projection from a state of $S \,[e]\, T$ to a state of S. It can be shown that if τ is a maximal computation of $S \,[e]\, T$ where e is *true* at all states, then $f(\tau)$ is a maximal computation of S. To complete the proof, let ω be a maximal computation of $S \,[e]\, T$ such that $P \wedge e$ holds at each state. By the antecedent (P **secures** Q **in** S), the maximal computation $f(\omega)$ therefore has a suffix where Q holds at each state. Therefore ω has a suffix where Q holds at each state.

∎

Hierarchical Composition: Let τ be an arbitrary maximal computation of $T; S$. The proof obligation is to show that if P holds at each state in τ, then there is a suffix in which R holds at each state. We show this in two steps. First, τ can be partitioned into a prefix and suffix so that Q holds at each state in the suffix. Second, this suffix is a maximal computation which has, in turn, a suffix in which R holds at each state. For the remainder of the proof we assume P holds at each state in τ.

First step. A small sublemma is needed for this step: if τ contains three consecutive states (a,b,c), it is not the case that $(f(a), f(b))$ is a transition of S and $(g(b), g(c))$ is a transition of T. This sublemma can be proved by contradiction. From the sublemma it follows that τ can be partitioned into $\tau = \delta\omega$ so that $g(\delta)$ is a computation of T and $f(\omega)$ is a computation of S (note that δ or ω may be empty). Further, it is simple to show (by contradiction) that $g(\delta)$ is a maximal computation of T and $f(\omega)$ is a maximal computation of S. Since $g(\delta)$ is a maximal computation of T and P holds at each state in $g(\delta)$, then by the antecedent (P **secures** Q **in** T) there exists a suffix of $g(\delta)$ such that Q holds at each state in the suffix; consequently $(P \wedge Q)$ holds at each state in this suffix of $g(\delta)$. On account of $(P \wedge Q) \Rightarrow$ **idle**.T we conclude that $g(\delta)$ is finite. We now

consider two cases for δ, either δ is empty or δ is finite and non-empty. In case δ is empty, we claim that $(P \wedge Q)$ holds at the initial state of ω (recall that τ is assumed non-empty, hence ω is non-empty for this case). The claim can be proved by contradiction. In case δ is non-empty, we observe that $(P \wedge Q)$ holds at the final state of δ, as established above. In either case, we have established that τ has a suffix with an initial state satisfying $(P \wedge Q)$; call this suffix *rho*. From $(P \wedge Q) \Rightarrow$ **idle**.T and **input** Q **in** S, observe that $(P \wedge Q)$ holds at each state in $g(\rho)$, hence $(P \wedge Q)$ holds at each state in ρ, and therefore Q holds at each state in $f(\rho)$. (end of first step).

Second step. We assert that ρ, as constructed in the first step, is a maximal computation of $T;S$ and $f(\rho)$ is a maximal computation of S. This assertion can be proved by contradiction. Since Q holds at each state of $f(\rho)$ and Q **secures** R **in** S, $f(\rho)$ has a suffix in which R holds at each state. Consequently, ρ has a suffix in which R holds at each state. (end of second step).

∎

9.3 *Proofs of Case Study*

The proof obligation from section 6.1 is to show *true* **secures** *homebusy* **in** *Busy*. We prove this by showing *Busy* **self-stabilizes to** *home*, where *home* is a predicate satisfying *home* $\Rightarrow$ *homebusy*. Then, by the self-stabilization and weakening rules, our proof obligation is fulfilled.

A state of *Busy* can be represented by a string of integers: each integer in the string specifies a value for an x-variable and thus the state of a process. Consequently, referring to an integer in a string is equivalent to referring to a process. We find it convenient to use string expressions as shorthand for some state predicates. For instance, the expression ba^{n-1} denotes states that satisfy

$$(x_k = \mathrm{b}) \wedge (\forall i: k < i < k+n: x_i = \mathrm{a}).$$

Recall that we index x modulo n, so there are n states corresponding to the expression ba^{n-1}. In string expressions, we use uppercase letters for arbitrary strings. The notation |R| refers to the length of the string R. A string of unit length (that is, an individual item) may be denoted by a lowercase letter. We use addition in string expressions to denote integer addition. Thus, $\mathrm{b}^{n-1}(\mathrm{b}+1)$ depicts states in which $n-1$ processes have the value b, and the remaining process has

the value (b +1). For a non-empty string R, the notation *first*(R) denotes the first integer of R and *last*(R) denotes the last integer of R.

Several definitions in this section contain terms of the form $x_j \bmod n = k \bmod n$. We abbreviate such terms with the notation $x_j =_n k$. Throughout the proof we assume $n > 1$ to simplify the presentation.

To clarify references to actions of *Busy*, we annotate the actions of a process S_i as follows:

{rel} $\quad x_i =_n i \quad \rightarrow \quad x_i := x_i + 1$,

{prop} $\quad x_{i-1} > x_i \quad \rightarrow \quad x_i := x_{i-1}$.

Definition. For integers k and m, the sequence of x-variable values $[\, x_i : k \le i < k+m \,]$ is a *rising string* iff $(\forall\, i : k < i < k+m : x_{i-1} + 1 = x_i) \;\wedge\; x_k =_n (k+1)$.

Definition. For integers k and m, the sequence of x-variable values $[\, x_i : k \le i < k+m \,]$ is a *falling string* iff $(\forall\, i :: k < i < k+m : x_{i-1} \ge x_i)$.

Definition. For integer k the sequence of x-variable values $[\, x_i : k \le i < k+n \,]$ is a *centered string* iff $(\forall\, i, j :: x_i > x_j \;\Rightarrow\; (x_i \neq_n j \;\vee\; x_i = x_k))$.
Informally, for the state of *Busy* to satisfy the definition of centered string, the outcome of any sequence of **prop** actions creates at most one token -- a token at process x_k.

Definition. The *home* predicate is satisfied by a state of *Busy* iff the state is of the form pFU, where pFU is a centered string, F is a falling string, Up is a rising string, and
$|\mathrm{F}| > 0 \;\Rightarrow\; (\, last(\mathrm{Up}) \ge first(\mathrm{F}) \;\wedge\; last(\mathrm{F}) \ge first(\mathrm{Up}) \,)$.

Lemma. *home* $\Rightarrow$ *homebusy*.

Proof. The definition of the *homebusy* predicate consists of a disjunct that holds for exactly one process of *Busy*: either that process holds a token or holds a token after a **prop** action. Using the string representation pFU of the *home* predicate and expanding definitions of centered, rising, and falling strings, it follows that *first*(FU) corresponds to the process satisfying the definition of *homebusy*. ∎

Lemma. Any transition from a state satisfying *home* results in a state satisfying *home*.

Proof by structural induction. We examine the string pFU and the outcome of any transition. There are two cases for a transition, either a **rel** action or a **prop** action induces the transition. For a **rel** transition, no process in the rising string is eligible; the centered string condition implies that the **rel** transition is due to the first process in the falling string. It is straightforward to verify that this case satisfies *home*. For a **prop** transition, there are two possibilities. The transition either occurs in the falling string or at the first process in the rising string. A **prop** transition due to a process in the falling string yields a result for which it is easy to verify *home*'s satisfaction. A **prop** transition at *first*(Up) requires more detailed examination. Conditions for this case imply $|U| > 0$. There are subcases for $|F| = 0$ and $|F| > 0$. For each subcase *home* 's satisfaction can be verified.

∎

Lemma. In every computation of *Busy* there is a state satisfying

$$(\exists y ::\ x_y = (y+1)\ \wedge\ (\forall j ::\ j \neq_n y\ \Rightarrow\ x_y > x_j)).$$

Proof. Let τ be an arbitrary computation of *Busy*. We define F to be a function from a state of *Busy* to an integer:

$$F(c) = (\textbf{max}\ i:\ 0 \leq i < n:\ x_i(c))$$

where $x_i(c)$ denotes the value of the variable x_i at state c. Let $z = F(e)$ where e is the initial state of τ. We define G to be a function from a state of *Busy* to an integer:

$$G(c) = (\textbf{sum}\ i:\ 0 \leq i < n:\ z - x_i(c)).$$

To prove the lemma, we examine the sequence of integers ω obtained by applying F to each state of the sequence τ. The following three observations, which are provable from the definitions above and the actions of *Busy*, establish that ω has a prefix of the form $y^m(y+1)$:

(i) For any transition (r,s) in τ: $(F(r) = y\ \wedge\ F(s) = y)\ \Rightarrow\ (G(r) > G(s)\ \wedge\ G(s) \geq 0)$.

(ii) For any transition (r,s) in τ: $(F(r) = y\ \wedge\ G(r) = 0)\ \Rightarrow\ F(s) = (y+1)$.

(iii) At any state of *Busy* a transition is possible, therefore τ is infinite.

Having established that ω has a prefix $y^m(y+1)$, we examine the transition (p,q) where $F(p) = y$ and $F(q) = (y+1)$. This transition is induced by a **rel** action in incrementing x_k for some k satisfying $x_k = y$. The notation x_y is shorthand for specifying x_k where $k =_n y$; hence existence of the state q constitutes proof of the lemma.

∎

Definition. Let C be a string of integers identified with a sequence of processes. A *subcomputation* with respect to C is a computation such that the initial state satisfies C and wherein each transition corresponds to some action by a process in C with one restriction: there is no **prop** action by the first process in C.

Lemma. For any string C, every subcomputation with respect to C is finite.

Proof by induction on |C|.

(basis) We take $|C| = 1$ for the base case. By definition any subcomputation with respect to C contains only transitions due to **rel** actions of the single process in C. After one such transition no further actions are possible (end of basis).

(induction) Let $bB = C$ where $|B| > 0$, and consider an arbitrary subcomputation with respect to bB. The subcomputation contains at most one transition due to a **rel** action at process b. Therefore consider a suffix τ that contains no such **rel** transition. The suffix τ contains at most one **prop** transition due to the first process in B. Consider a suffix of τ that contains no such **prop** transition. This suffix satisfies the definition of a subcomputation with respect to B. By the inductive hypothesis, any subcomputation with respect to B is finite. ∎

Definition. The *homeward* predicate is satisfied by a state of *Busy* iff the state is of the form AC where $|AC| = n$, $|A| > 1$, $Ar^{|C|}$ satisfies *home* where $r = last(A)$, and C satisfies the following constraint. Every subcomputation τ with respect to C satisfies: at each state in τ all x-variable values of processes in C are smaller than any value in A.

Lemma. Every computation of *Busy* contains a state satisfying *homeward*.

Proof. Consider an arbitrary computation of *Busy*. By previous lemma, the computation contains a state satisfying the form $(y + 1)W$ such that $x_y = (y + 1)$ and $(y + 1)$ is the maximum x-value. Let τ be a suffix of the computation with $(y + 1)W$ as initial state. The suffix τ has a prefix (possibly empty) that is a subcomputation with respect to W. Let μ be the longest prefix of τ that is a subcomputation with respect to W. Each state of μ is of the form $(y + 1)W'$, from which no transition due to an action at x_y is possible. Therefore, the state in τ immediately following μ is due to a **prop** action at x_{y+1} with the resulting state of the form $(y + 1)(y + 1)U$, which satisfies the definition of *homeward*. ∎

Lemma. Any transition from a state satisfying *homeward* results in a state satisfying *homeward*.

Proof. We examine the string AC from *homeward*'s definition and the outcome of any transition. The case $|C| = 0$ satisfies *home* and a previous lemma establishes the invariance of *home*. Therefore we assume $|C| > 0$ for the remainder of the proof. There are three forms for the outcome of a transition: either A′C, AC′, or A″C″ where $|A''| = |A| + 1$. The first form A′C is due to an action within A but not due to a **prop** action by the first process of A because C is non-empty and all values in C are smaller than any value in A. Consequently A′C satisfies *homeward*. The second form AC′ is due to an action within C but not due to a **prop** action by the first process of C. By *homeward*'s definition and the fact that C′ is the result of a subcomputation with respect to C, the result AC′ satisfies *homeward*. The third form A″C″ is due to a **prop** action by the first process in C, and also satisfies *homeward*. ∎

Lemma. Any computation containing a state of the form AC where $|C| > 0$ and AC satisfies *homeward* also contains a state of the form A″C″ where A″C″ satisfies *homeward* and $|C| > |C''|$.

Proof. The proof of the previous lemma outlines the case analysis needed for this proof. Observe that the forms A′C and AC′ are due to subcomputations. Consequently, such transitions appear a finite number of times in a computation. The computation therefore contains a state of the form A″C″ satisfying the conclusion. ∎

Lemma. Every computation of *Busy* contains a state satisfying *home*.

Proof. Every computation contains a state satisfying *homeward*; such a state is of the form AC. Using the previous lemma, it follows by induction on |C| that a state of the form A″C″ is in the computation where $|C''| = 0$. This state satisfies *home*. ∎

The preceeding lemmas show that *Busy* self-stabilizes to *home*. The definition of *home* can also be used to show that each process of *Busy* has a token infinitely often in any computation, thereby justifying the name "busy."

The proof obligation from section 6.2 is to show *true* **secures** *homelazy* **in** *Lazy*. The *homelazy* predicate and the *homebusy* predicate have the same structure; the proof of *true* **secures** *homebusy* consists of arguments about **prop** and **rel** actions in computations; *Busy* and *Lazy* have identically structured **prop** actions, while the **rel** action of *Lazy* has a stronger precondition. The stronger

precondition does not complicate the analysis of *Lazy*. The definitions and lemmas used to prove *true* **secures** *homebusy* **in** *Busy* can, with small modifications, be applied to show *true* **secures** *homelazy* **in** *Lazy*. The structure of the *Lazy* version of the *home* predicate can also be used to prove that if no process needs a token then any computation is finite, thereby justifying the name "lazy."

References

[1] G. M. Brown, M. G. Gouda, and C. L. Wu, "Token Systems that Self-Stabilize," *IEEE Transactions on Computers 38*, (6 1989).

[2] K. Mani Chandy and Jayadev Misra, *Parallel Program Design: a Foundation,* Reading, Massachusets: Addison-Wesley Publishing Company, 1988.

[3] Edsger W. Dijkstra, "Self-stabilizing Systems in Spite of Distributed Control," *CACM 17*, (11 1974), pp. 643-644.

[4] Ted Herman, Ph. D. Dissertation, Dept. of Computer Sciences, University of Texas at Austin, in preparation, 1989.

[5] C. A. R. Hoare, "Communicating Sequential Processes," *Comm. ACM 21*, (8 Aug 1978).

[6] S. Lam and A. Shankar, "Specifying Implementations to Satisfy Interfaces: A State Transition System Approach," presented at the 26th Lake Arrowhead Workshop, Sept. 1987.

[7] L. Lamport, "Specifying Concurrent Program Modules," *ACM TOPLAS 5*, (2 April 1983).

[8] N. Lynch, "I/O Automata: A Model for Discrete Event Systems," *Proc. of 22nd Annual Conf. on Information Sciences and Systems*, 1988.

[9] R. Milner, *A Calculus of Communicating Systems,* LNCS 92, Springer-Verlag, New York, 1980.

[10] A. Pnueli, "Linear and Branching Structures in the Semantics and Logics of Reactive Systems," *Proc. of the 12th ICALP*, 1985.