The Maximality of Unhygienic Dining Philosophers

Matthew Lang and Paolo A. G. Sivilotti Department of Computer Science and Engineering
The Ohio State University
Columbus, Ohio 43210-1277
Email: {langma,paolo}@cse.ohio-state.edu

Abstract

The behaviors exhibited by a correct program are a subset of those allowed by its specification. A program is said to be maximal if, in addition to being correct, it can exhibit *any* of the behaviors permitted by its specification. Maximality is useful for design, as it eliminates trivial or degenerate solutions; for performance as it requires solutions to permit maximum concurrency; and for testing as it exposes mistakes in layered implementations. Unfortunately, maximality is not compositional: The composition of two maximal programs is not guaranteed to be maximal. This observation poses a challenge for the modular verification of maximality in composed systems. In this paper, we present a strategy for proving the maximality of composed systems in a modular manner. As an example, we consider a classic scheduling problem, dining philosophers. We present the unhygienic algorithm and prove both its correctness and maximality. This proof establishes the unhygienic solution as the first known maximal dining philosophers algorithm.

I. Introduction

A specification describes a set of possible program executions; a program implementing a specification is correct if the set of possible executions of the program is contained within the specification. Maximality is a stronger notion. A program is *maximal* if it is (i) correct, and (ii) the set of behaviors permitted by the specification is *contained within the possible behaviors of the program*. That is, the program can exhibit *all* the behaviors permitted by the specification.

Maximal programs are important because they eliminate trivial solutions to problems—those that limit concurrency or otherwise restrict behavior permitted by a specification. Maximal programs are also important for testing component-based systems because they prevent a component implementation from providing unnecessarily deterministic behavior and, in this way, masking errors in its clients.

For example, consider a bag data structure with two operations, *put* and *get*. Informally, the specification for this data structure states that the *put* operation adds objects to the bag and the *get* operation returns an arbitrary object from the bag. A FIFO queue is clearly a correct implementation of this specification. However, this implementation restricts the behavior permitted by the specification: The i^{th} *get* always returns the object placed in the bag by the i^{th} *put*. The FIFO queue implementation, therefore, is not maximal with respect to the specification for a bag.[1] Using this implementation to model an unordered communication channel would restrict the possible behaviors of the composed system.

Unfortunately, the maximality of concurrent programs is not compositional. That is, the parallel composition of two maximal programs is not, itself, guaranteed to be maximal. The current technique for proving the maximality of a UNITY-like program involves whole-program analysis [4]. In this paper, we extend this technique to prove the maximality of composed programs in a modular way. Essentially, the proof technique involves adding requirements to a component's environment that permit the whole-program proof obligations to be dispatched without the implementation particulars of the environment.

The dining philosophers problem, or *diners* for short, is a classic synchronization problem for distributed systems. Solutions to this problem are inherently compositional. Not only are there multiple dining processes, but each process consists of two parts: a user that requests a resource and a conflict resolution layer that grants the resource. To illustrate our method for modular proofs of maximality, we describe the *unhygienic* algorithm for diners. We prove that this algorithm is both a correct and maximal implementation of the conflict resolution layer.

[1] It is, however, maximal with respect to a different specification, namely the specification for a FIFO queue. The maximality of an implementation, therefore, is relative to a particular specification.

The paper is organized as follows. Section II provides background, including the model of computation, the whole-program technique for proving maximality, and the diners problem. Section III exposes the non-compositional nature of maximality. Section IV describes the unhygienic algorithm and provides an informal proof of its correctness. Section V illustrates our method for proving the maximality of composed programs by proving the maximality of the unhygienic algorithm. Sections VI and VII conclude the paper by examining the limitations of our approach, pointing to related work, and indicating directions for future research.

II. Background

A. The Programming Model

We use a UNITY-style notation for writing programs and reasoning about their properties [1], [3], [7]. Programs are sets of guarded commands of the form $A : \gamma \longrightarrow \alpha$ where A is a label, γ is a guard which is a predicate on program variables, and α is a command consisting of assignments to program variables. An action whose guard is true is said to be enabled. Each program implicitly includes the action $skip$, which does nothing. Execution of a UNITY program is an infinite sequence of states, where each pair of successive states is the result of the execution of a program action. The result of executing an enabled action is the result of the assignments in the actions command. The result of executing an action that is not enabled is no change in state. A program execution is weakly fair: each action is executed infinitely often.

For program properties, the basic safety operator is **next**. The property X **next** Y says that every state that satisfies the predicate X is immediately followed by a state that satisfies the predicate Y. Since every program contains $skip$, this is only possible if $X \Rightarrow Y$. Similarly, **stable** X requires that every state that satisfies X is immediately followed by a state that also satisfies X. Hence, once X holds, it continues to hold. An invariant property is true initially and stable. Finally, X **unless** Y says that X cannot become false without Y being true. Formally, X **unless** $Y \equiv (X \wedge \neg Y)$ **next** $(X \vee Y)$.

A basic progress operator is **transient**. The property **transient** X says that a program has at least one action that, when executed in a state where X holds, results in a state where $\neg X$ holds. Thus, $\neg X$ is guaranteed to hold infinitely often. Any transient property of a program is also a property of a composed system that contains that program. The progress property $X \rightsquigarrow Y$ (X leads-to Y) requires that if X holds at any point in a computation, Y holds at a later (or the same) point.

When free variables appear in these properties, they are understood to be implicitly universally quantified. So, for a program with variable x, the property **stable** $x = k$ should be read as $(\forall k :: \textbf{stable } x = k)$; that is, x never changes value.

B. Proving Maximality

A program P is correct with respect to a specification S if the set of possible executions of P, $|P|$, is contained in the set of traces that satisfy S, $|S|$. That is, $|P| \subseteq |S|$. On the other hand, a program P is *maximal* with respect to a specification, S, if P is correct and $|P| = |S|$ [7], [4], [5]. The notion of maximality is similar to bisimulation [6], [8] but the former relates program texts and specifications while the latter involves artifacts with similar mathematical representations, such as two Kripke structures.

The proof technique for establishing maximality consists of two steps. The first step is to show that an arbitrary trace $\sigma \in |S|$ is a possible execution of an instrumented version P' of P. The second step is to show that every fair execution of P' corresponds to a fair execution of P.

P' is constructed from P by (i) adding new variables, (ii) adding assignments to new variables within existing actions, (iii) adding guards to existing actions, and (iv) adding actions that mention only new variables. The new variables include *chronicles* that encode the arbitrary trace $\sigma \in |S|$ and auxiliary variables that encode, for example, the current point, i, in the trace. Assignments in P' may modify auxiliary variables but not chronicles. These modifications ensure that every safety properties of P is also a property of P'.

To show that σ is a possible execution of P', it suffices to show that for every value of i, the value of program variables of P are equal to σ_i.

The second obligation, that every fair execution of P' corresponds to a fair execution of P, requires showing that (i) each additional guard is infinitely often true and (ii) the truth of each additional guard is preserved by the

execution of every action of P' except the action it guards. These two properties ensure that each action of P' is executed infinitely often in a state where the additional guard is true.

This proof technique is illustrated in [7] using the task scheduling problem. This problem consists of a set of tasks and a conflict relation between tasks. Given that executing tasks eventually terminate, algorithms that solve the task scheduling problem schedule tasks for execution such that no pair of tasks in the conflict relation execute concurrently and each task is executed infinitely often. The task scheduling problem is closely related to diners, and its maximal solution is the basis for the unhygienic diners algorithm. The key difference is that the task scheduling solution is a single program, whereas a diners solution is a part of a composed program.

C. The Dining Philosophers Problem

The diners problem [2] is a classic resource allocation problem. Originally framed as a set of philosophers arranged around a table, the problem has been generalized to an arbitrary graph in which nodes correspond to processes and edges represent a conflict for a shared resource.

A process can be in one of three states: *thinking*, *hungry*, or *eating*. A client process (the user) controls transitions from thinking to hungry and eating to thinking while an arbitration process (the conflict resolution layer) controls the transition from hungry to eating.

The user process guarantees that eating is finite. The problem is to design an implementation for the conflict resolution layer such that the composed system satisfies the following properties:

- Safety: No two neighboring processes are in the eating state simultaneously.

  invariant $(\forall u, v : u \textbf{ nbr } v : \neg(u.state = eating \wedge v.state = eating))$

- Progress: A hungry process eventually eats.

  $u.state = hungry \rightsquigarrow u.state = eating$

One strategy adopted by several algorithms to solve the diners problem is to associate a token, a fork, with each edge in the conflict graph. A process is required to hold all its forks in order to eat. Since forks are neither created nor destroyed, this strategy ensures the safety part of the specification is satisfied.

III. The Non-Compositional Nature of Maximality

Maximality properties, much like non-trivial progress properties, are noncompositional. That is, given two programs P_0 and P_1 that maximally implement the specifications S_0 and S_1 respectively, there is no guarantee that their composition, $P_0 \parallel P_1$, is maximal with respect to $S_0 \parallel S_1$.

For example, consider the specification S_0:

$x = k$ **next** $|x - k| \leq 1$

transient $x = k$

This specification says that x changes value infinitely often and that each change of value is by at most one.

An implementation of this specification is given by program P_0:

initially $a = 1$

assign

A_0 $x := x + a$

B_0 $a := -a$

This implementation is maximal since any sequence of values admitted by the specification can be generated through the appropriate choices of which action to execute. For example, the sequence in which x never decreases is formed by always selecting B_0 twice between successive choices of A_0.

Now consider the specification S_1:

$x = k$ **next** $|x - k| \leq 2$

transient $x = k$

Like S_0, this specification also says that x changes value infinitely often. Unlike S_0, however, S_1 allows x to change value by at most two.

An implementation of this specification is given by program P_1:

initially $b = 1$
assign
A_1 $x := x + b$
B_1 $b := -b$
C_1 $b := 3 - |b|$

This program is maximal with respect to S_1.

The specification for the composed system ($S_0 \parallel S_1$) is given by:

$x = k$ **next** $|x - k| \leq 2$
transient $x = k$

Any correct implementation of S_0, when composed with any correct implementation of S_1, is guaranteed to satisfy this specification. Furthermore, this is the strongest specification for which this claim can be made.

Indeed, the composition of programs, $P_0 \parallel P_1$ does satisfy this composed specification: x can change value by at most 2 and x changes value infinitely often. The composed program, however, is not maximal. Consider the sequence of values in which whenever x changes value, it changes by 2 (e.g., $\langle 2, 4, 6, 8, 10, 12, \ldots \rangle$) This sequence is permitted by the composed specification, but can not be generated by the composed program. Weak fairness requires that all actions be chosen infinitely often. However, whenever action A_0 is chosen, x changes value by 1. Thus, every trace generated by the composed program must include an infinite number of points at which x changes value by 1, and hence the program is not maximal.

IV. The Unhygienic Dining Philosophers Algorithm

In this section, we first present a classic solution to the dining philosophers problem: the hygienic algorithm [1]. This algorithm uses fork tokens to ensure mutual exclusion and a dynamic priority ordering to ensure progress. Through a series of generalizations, we transform this algorithm into one that permits repeated overtaking. This new algorithm is the *unhygienic* dining philosophers algorithm. This algorithm has appeared in [10] and [9], however this paper is the first proof of its maximality.

A. Background: Hygienic Dining Philosophers

The hygienic dining philosophers algorithm [1] is a solution to the generalized dining philosophers problem. Each pair of neighbors in the conflict graph shares a single fork. A process is required to hold all its forks in order to eat, thus ensuring mutual exclusion.

To ensure progress, philosophers are assigned relative priorities. Each edge in the conflict graph is directed from the lower priority neighbor to the higher priority neighbor, and initial priorities are assigned so that the directed graph is acyclic. The priority graph remains acyclic since priorities only change when a philsopher eats, at which time it lowers its priority below all its neighbors. One way to satisfy this requirement of acyclicity is to assign each philosopher a height so that neighbors do not have the same height. A philosopher changes height only by eating and when it eats it lowers its height below all its neighbors.

When a philsopher becomes hungry, it requests all of its missing forks. Conversly, when a philosopher receives a request for a fork, it releases the fork only if it is not eating and the requesting neighbor has higher priority.

Figure 1 shows a UNITY program that implements the conflict-resolution layer of this algorithm. The program in Figure 1 is composed with a user program that controls state transitions from thinking to hungry, and from hungry to eating. Throughout this paper, we define the predicate $u.t$ ($u.h$, $u.e$) to be $u.state = thinking$ ($hungry$, $eating$ respectively).

We use the term *low fork* to refer to a fork that is shared with a lower-priority neighbor. The same fork is a *high fork* for that neighbor. The direction of priority for an edge is encoded on a fork as the difference in heights between the processes that share the fork. In the original presentation of this algorithm, high forks were termed *dirty*, since

Program $hygienic_u$
initially $u.t$
$(\forall v : u \textbf{ nbr } v : u.ht \neq v.ht)$
$(\forall v : u \textbf{ nbr } v : fork(u,v) = u \equiv u.ht > v.ht)$
always $sendreq(u,v) \equiv fork(u,v) = v \wedge rf(u,v) = u \wedge u.h$
$sendfork(u,v) \equiv fork(u,v) = u \wedge rf(u,v) = u \wedge \neg u.e \wedge u.ht < v.ht$
$u.mayeat \equiv (\forall v : u \textbf{ nbr } v : fork(u,v) = u \wedge (u.ht > v.ht \vee rf(u,v) = v))$
assign
E_u : $u.h \wedge u.mayeat \longrightarrow u.state := eating$
$\| \; u.ht := (\textbf{Min}\, v : u \textbf{ nbr } v : v.ht) - 1$
H_u : $sendreq(u,v) \longrightarrow rf(u,v) := v$
R_u : $sendfork(u,v) \longrightarrow fork(u,v) := v$

Fig. 1. Hygienic Dining Philosophers Algorithm

all forks are high forks after a philosopher eats. Conversly, low forks were termed *clean*. Since only high (dirty) forks are released to a neighbor, and since a fork switches from high to low when received by that neighbor, all forks arrive as clean forks. This observation motivates the choice of the name, hygienic, for this algorithm.

B. Generalization: Weakening the Invariant

One invariant of the hygienic algorithm is that thinking philosophers never hold low forks. This property is true initially since all forks are initially high forks. It is maintained when a philsopher begins thinking since all forks became high forks when it ate. Also, it is maintained when (low) forks arrive since a thinking philosopher never acquires forks.

Because of this invariant, the rule for releasing a fork is simply expressed: a philosopher never releases a low fork. This rule, however, is stronger than needed to prove correctness. To guarantee progress, it suffices to require that a *hungry* philosopher never releases a low fork. That is, the $sendfork$ predicate can be rewritten as follows:

$$sendfork(u,v) \equiv fork(u,v) = u \wedge rf(u,v) = u \wedge (u.t \vee (u.h \wedge u.ht < v.ht))$$

This version is equivalent to the original $sendfork$ predicate given the invariant that thinking philosophers never hold low forks.

If we relax the invariant, however, the algorithm with the new formulation is still correct. That is, we remove the following predicate from the initial conditions:

$$(\forall v : u \textbf{ nbr } v : fork(u,v) = u \equiv u.ht < v.ht)$$

Without this requirement, the initial placement of a fork is arbitrary, so thinking philosophers may hold low forks. Such a philosopher, however, does not prevent a lower priority hungry philosopher from eating without it itself becoming hungry.[2]

With this generalization, low (clean) forks can be released, specifically by a thinking process. Because a fork switches from being high to being low when received by that neighbor, such forks arrive as dirty forks. This observation motivates the choice of the name, unhygienic, for this algorithm.

C. Generalization: Increasing Nondeterminism

Having modified the algorithm to permit thinking philosophers to hold low forks, it is no longer necessary for eating to result in a process being lowered below all of its neighbors. Instead, it suffices for eating to result in *some* lowering of height. Thus, the eating action is modified to lower a process by a nondeterministic amount.

$$u.ht :=? \textbf{ st } u.ht < u.ht_{pre} \wedge (\forall v : u \textbf{ nbr } v : u.ht \neq v.ht)$$

[2]We note that this algorithm stabilizes to the invariant that a low fork is not held by a thinking philosopher so long as either neighbor that shares the fork becomes hungry at least once. Thus, the only case where stabilization to the original invariant does not occur is if neither neighbor becomes hungry at least once, in which case the fork is never requested.

This generalization allows a hungry process to be overtaken repeatedly by a neighbor. In the hygienic algorithm, once a process u becomes hungry, no neighbor, v, of u can eat more than once. After eating, v is lowered below u, so it must relinquish the shared fork. In the unhygienic algorithm, however, v is lowered by some amount, but not necessarily below u. At that point, v is thinking and holding a low fork. Although it might release the fork (action R_v), it is not required to do so immediately. If, instead, v becomes hungry again, then the fork is not released and v will eat a second time while u remains hungry. Thus, the unhygienic algorithm permits finite, but unbounded, overtaking of hungry philosophers. The algorithm is summarized in Figure 2.

Program $unhygienic_u$
initially $u.t$
$(\forall v : u \textbf{ nbr } v : u.ht \neq v.ht)$
always $sendreq(u,v) \equiv fork(u,v) = v \wedge rf(u,v) = u \wedge u.h$
$sendfork(u,v) \equiv fork(u,v) = u \wedge rf(u,v) = u \wedge (u.t \vee (u.h \wedge u.ht < v.ht))$
$u.mayeat \equiv (\forall v : u \textbf{ nbr } v : fork(u,v) = u \wedge (u.ht > v.ht \vee rf(u,v) = v))$
assign
E_u : $u.h \wedge u.mayeat \longrightarrow u.state := eating$
$; u.ht :=?$ **st** $u.ht < u.ht_{pre} \wedge (\forall v : u \textbf{ nbr } v : u.ht \neq v.ht)$
H_u : $sendreq(u,v) \longrightarrow rf(u,v) := v$
R_u : $sendfork(u,v) \longrightarrow fork(u,v) := v$

Fig. 2. Unhygienic Dining Philosophers Algorithm

D. Proof of Correctness

We provide an sketch of the proof of correctness of the unhygienic algorithm.

The safety requirement, that no two neighboring processes eat simultaneously, is ensured by the invariant that an eating process holds all its forks. The proof is identical to that for the hygienic algorithm.

The progress requirement, that a hungry process eventually eats, is ensured by identifying a metric for a hungry process. This metric must be bounded below and must be guaranteed to decrease unless the process eats. In the hygienic algorithm, this metric (for a process u) is the sum of the number of higher processes and the number of higher thinking processes. More formally, let $u.above$ be the set of processes reachable from u. Recall that edges are directed up, so $v \in u.above$ means that there is a path consisting only of edges to higher priority processes from u to v. Then $u.m$ is defined by:

$$u.m = |\{ v :: v \in u.above \}| + |\{ v :: v \in u.above \wedge v.t \}|$$

This metric is nonincreasing. Furthermore, the metric is guaranteed to decrease for a hungry process by the following argument. If u is hungry, there must a process above u (or u itself) that is both hungry and has no higher priority hungry neighbors. This process is guaranteed either to eat eventually (in which case the number of processes above u decreases), or to have one of its higher neighbors become hungry (in which case the number of thinking processes above u decreases). In either case, $u.m$ decreases.

For the unhygienic algorithm, the argument is similar. The metric for a hungry process is provided by:

$$u.m = (\sum v : v.ht > u.ht : v.ht) + (\sum v : v.ht > u.ht \wedge v.t : v.ht)$$

In this case, the sum of heights is taken over all processes that are higher than u, which is a superset of $u.above$. As before, this metric is nonincreasing. Also, as before, the metric is guaranteed to decrease for a hungry process. If u is hungry, there must a process above u (or u itself) that is both hungry and has no higher priority hungry neighbors. This process is guaranteed either to eat eventually (in which case the sum of heights of processes above u decreases), or to have one of its higher neighbors become hungry (in which case the number of thinking processes above u decreases). In either case, $u.m$ decreases.

V. Maximality of the Unhygienic Algorithm

Because the unhygienic algorithm permits overtaking of hungry neighbors, it allows for executions that are not possible for the hygienic algorithm. In this section, we prove that there are no executions that satisfy the diners specification that are not possible for unhygienic algorithm. That is, we show that this algorithm is maximal.

Maximality is not compositional, so proving the maximality of a single component, such as the conflict resolution layer, in isolation (i.e., without the user process) is not enough. Conversely, proving the maximality of the conflict resolution layer composed with a particular user implementation is too specific to that particular implementation. Instead, here we postulate a set of properties that the user process satisfies and then carry out the proof of maximality based on these properties. These properties are in addition to the usual requirements on $user$ needed for correctness of the composed system, for example that eating is finite.

We begin by defining the structure of a maximal trace satisfying the diners problem specification. We then modify $unhygienic$, adding guards and variables to form a constrained program $unhygienic'$. We then state the additional requirements placed on the $user$ process. These additional requirements ensure that (i) $user$ is maximal and (ii) $user$ can be constrained in a particular way to establish this maximality.

Finally, we prove the maximality of the composed system of constrained programs. For any maximal trace, the constrained system is shown to compute the execution sequence described by the trace. Furthermore, any fair execution of the constrained system corresponds to a fair execution of the original system.

In our presentation of the unhygienic algorithm in Section IV, process height corresponded directly to priority. That is, if a process had greater height, it also had higher priority. Thus, after eating a process lowered its priority by lowering its height. In this section, however, we reverse this priority ordering. That is, if a process has greater height, it has *lower* priority. Thus, after eating a process lowers its priority by *raising* its height. We make this change for clarity in the proof of maximality, where process height corresponds to a point in the trace, and hence increases over the course of the computation. We will avoid confusion by always explicitly distinguishing the two. For example, a neighbor will be referred to as a "higher priority neighbor" rather than simply a "higher neihbor".

A. A Maximal Trace of $user \parallel unhygienic$

Let S be a sequence of tuples representing the states of processes in an execution satisfying the specification of the dining philosopher's problem. Let S_i denote the i^{th} tuple in the sequence and let $S_{u,i}$ denote the state of process u at step i ($S_{u,i} \in \{t, h, e\}$ representing the thinking, hungry, and eating states respectively). We consider only stutter-free sequences. That is, each tuple in the sequence differs from the previous in at least one element, *unless* the execution is in a state of quiescence (every process remains in the thinking state forever).

The following properties characterize any trace, S, that satisfies the diners specification. These properties apply to all processes u and all points i in the computation.

- At each step in the computation, at most one process changes state.

$$S_{u,i} \neq S_{u,i+1} \Rightarrow (\forall v : u \neq v : S_{v,i} = S_{v,i+1}) \tag{S0}$$

- The trace is stutter-free, except in the state of quiescence.

$$S_i = S_{i+1} \Rightarrow (\forall v, j : j \geq i : S_{v,j} = t) \tag{S1}$$

- Processes are initially thinking, and valid state transitions must loop through thinking, hungry, and eating states.

$$S_{u,0} = t \tag{S2}$$
$$S_{u,i} = t \Rightarrow (S_{u,i+1} = t \vee S_{u,i+1} = h) \tag{S3}$$
$$S_{u,i} = h \Rightarrow (S_{u,i+1} = h \vee S_{u,i+1} = e) \tag{S4}$$
$$S_{u,i} = e \Rightarrow (S_{u,i+1} = e \vee S_{u,i+1} = t) \tag{S5}$$

- Eating is finite.

$$S_{u,i} = e \Rightarrow (\exists j : j > i : S_{u,j} = t)$$

- Neighbors do not eat simultaneously and a hungry process eventually eats.

$$(\forall v : u \textbf{ nbr } v : \neg(S_{u,i} = e \wedge S_{v,i} = e)) \quad \text{(S6)}$$

$$S_{u,i} = h \Rightarrow (\exists j : j > i : S_{u,j} = e) \quad \text{(S7)}$$

B. The Constrained Conflict Resolution Layer

In the constrained program, we introduce four objects not found in the original: (i) the input trace S, (ii) a variable p to indicate the current point in S, (iii) a function $u.next$ to compute the next point at which process u begins to eats, and (iv) a predicate $u.done$ to denote whether or not process u eats again after the current point in the computation.

Intuitively, we will use these to assign a process's height to be the next point in the computation when it begins to eat. Then at that point in the computation it will be the highest priority hungry process among its neighbors. Figure 3 shows the constrained program. Note that it is not an executable program— $u.done$ is a predicate over an infinite trace. Note also that the inequalities in $sendfork$ and $mayeat$ have been reversed from Section IV reflecting the reversal of the priority relation.

Program $unhygienic'_u$

initially

$u.t \wedge p = 0$

$(\forall v : u \textbf{ nbr } v : u.ht \neq v.ht)$

$\neg u.done \Rightarrow u.ht = u.next$

$u.done \Rightarrow (\forall v : u \textbf{ nbr } v : S_{v,u.ht} \neq e \wedge S_{v,u.ht-1} \neq e)$

always

$sendreq(u,v) \equiv fork(u,v) = v \wedge rf(u,v) = u \wedge u.h$

$sendfork(u,v) \equiv fork(u,v) = u \wedge rf(u,v) = u \wedge (u.t \vee (u.h \wedge u.ht > v.ht))$

$u.mayeat \equiv (\forall v : u \textbf{ nbr } v : fork(u,v) = u \wedge (u.ht < v.ht \vee rf(u,v) = v))$

$u.done \equiv (\forall i : i \geq p : S_{u,i} \in \{e,t\})$

$$u.next = \begin{cases} (\textbf{Min}\, i : i \geq p \wedge S_{u,i} = h \wedge S_{u,i+1} = e : i+1) & \text{if } \neg u.done \\ p+1 & \text{otherwise} \end{cases}$$

assign

E'_u : $u.done \vee u.ht = p+1 \longrightarrow$
$u.h \wedge u.mayeat \longrightarrow u.state := eating$
$; p := p+1$
$; u.ht := u.next$

H'_u : **true** $\longrightarrow$
$sendreq(u,v) \longrightarrow rf(u,v) := v$

R'_u : **true** $\longrightarrow$
$sendfork(u,v) \longrightarrow fork(u,v) := v$

A : $S_p = S_{p+1} \longrightarrow p := p+1$

Fig. 3. The Constrained Unhygienic Algorithm $unhygienic'_u$

In this program, $u.next$ is well-defined since $\neg u.done \Rightarrow (\exists i : i \geq p : S_{u,i} = h \wedge S_{u,i+1} = e)$

$\neg u.done$
$\Rightarrow$ { def'n of $u.done$ }
$(\exists i : i \geq p : S_{u,i} = h)$
$\equiv$ { (S7) }
$(\exists i : i \geq p : S_{u,i} = h) \wedge (\exists j : j > i : S_{u,j} = e)$
$\equiv$ { (S4) }
$(\exists i : i \geq p : S_{u,i} = h \wedge S_{u,i+1} = e)$

The following are properties of $unhygienic'_u$, for all points i in the computation. These properties all follow directly from the program text.

- The trace is not changed.

  constant S (H0)

- The current point in the computation advances by at most one.

  $p = k$ **unless** $p = k + 1$ (H1)

- A process's height is equal to $u.next$, so long as it has not eaten for the last time.

  invariant $\neg u.done \Rightarrow u.ht = u.next$ (H2)

  $u.ht = k$ **unless** $p = k$ (H3)

In addition, the following properties can be easily shown.

- Tranquility is stable.

  stable $u.done$ (H4)

  Proof: Once $u.done$ holds, it can only be invalidated by S changing such that for some $i \geq p$, $S_{u,i} = h$ or by p decreasing. By property (H0), S is constant and by property (H1), p is nondecreasing. Therefore, once $u.done$ holds, it continues to hold. □

- A process's height is equal to the point in the computation at which it will next eat, so long as it has not eaten for the last time.

  invariant $\neg u.done \Rightarrow (S_{u,u.ht-1} = h \wedge S_{u,u.ht} = e)$ (H5)

  Proof:

$$
\begin{array}{ll}
 & u.ht = k \ \wedge \ \neg u.done \\
\equiv & \quad \{ \ \text{(H2)} \ \} \\
 & u.ht = u.next = k \ \wedge \ \neg u.done \\
\Rightarrow & \quad \{ \ \text{def'n } u.next = (\mathbf{Min}\, i : i \geq p \ \wedge \ S_{u,i} = h \ \wedge \ S_{u,i+1} = e : i + 1) \ \} \\
 & u.ht = k \ \wedge \ (S_{u,k-1} = h \ \wedge \ S_{u,k} = e)
\end{array}
$$

□

C. The Constrained User Program

At this point, we will state our assumptions on the $user_u$ process. Specifically, assume that $user_u$ can be constrained to produce $user'_u$ which has the following properties, for all points i in the computation.

- The trace is not changed.

  constant S (U0)

- The state corresponds with the current point in the trace.

  invariant $u.state = S_{u,p}$ (U1)

- The user program does not change the current point in the computation when the process is hungry or does not change state in the trace.

  stable $p = k \wedge u.h$ (U2)

  stable $p = k \wedge S_{u,k} = S_{u,k+1}$ (U3)

- The current point in the computation advances by at most one.

  $p = k$ **unless** $p = k + 1$ (U4)

- The trace dictates transitions out of the thinking and eating states.

  transient $S_{u,p} = t \wedge S_{u,p+1} = h$ (U5)

  transient $S_{u,p} = e \wedge S_{u,p+1} = t$ (U6)

- Guards added to the constrained user program are infinitely often true and, once one of these new guards becomes true it remains true until the corresponding action executes.

In addition, we require that $user'_u$ be produced by only adding new variables which are not program variables of $unhygienic_u$, assignments to new variables, and additional guards referencing new and program variables. Finally, if $user'_u$ replaces random assignments in $user_u$ with assignments referencing new and program variables, the assigned value satisfies the predicate on the random assignment in $user_u$. This ensures that the safety properties of $user_u$ are maintained in $user'_u$.

D. Proof of Maximality

To prove the maximality of our solution we show that S is a possible execution of $unhygienic \parallel user$. The proof is broken into several lemmas. Lemmas 1 and 3 establish that every fair execution of $unhygienic' \parallel user'$ compute the same states as S. Lemmas 4, 5, and 6 establish that every fair execution of $unhygienic' \parallel user'$ is a fair execution of $unhygienic \parallel user$.

Lemma 1: At each step in the computation, each process's state is the same as S:

invariant $u.state = S_{u,p}$

Proof: First we show that the invariant holds initially. From the program text, all processes are initially thinking and $p = 0$. By (S2), $S_{u,0} = t$ for all u. So, initially, the invariant holds.

Next we show that every action of $unhygienic'_u$ preserves the invariant. The only two actions that modify $u.state$ or p are E'_u and A. When E'_u is enabled, $(u.ht = p+1 \vee u.done) \wedge u.h$ holds. From the inductive hypothesis and $u.h$, however, we know $S_{u,p} = h$ and therefore $\neg u.done$. So, if E'_u is enabled, $\neg u.done \wedge u.ht = p+1$. By (H5), $S_{u,p+1} = e$. Since action E'_u both increments p and assigns the state to be eating, the invariant holds after it executes. When A is enabled, $S_p = S_{p+1}$. Since A increments p and does not change $u.state$, the invariant holds after it executes. Thus, $u.state = S_{u,p}$ is an invariant of $unhygienic'_u$.

By (U1), $u.state = S_{u,p}$ is also an invariant of $user'_u$. Therefore, it is an invariant of the composed program. □

Next, we present two observations and a lemma that will be used in the proof of Lemma 3.

Observation 1: $(\neg u.done \wedge u.ht = p+1) \Rightarrow (\forall v : u \textbf{ nbr } v \wedge v.ht \leq u.ht : v.done)$

Proof: By contradiction. Given $\neg u.done \wedge u.ht = p+1$, assume v is an arbitrary neighbor of u such that $v.ht \leq u.ht$ and $\neg v.done$. By (H2) and assumption $\neg v.done$, $v.ht = v.next$ and furthermore $v.ht > p$ (as $v.next > p$). But $u.ht = p+1$ and each process has a distinct height among its neighbors (see Lemma 6). $v.ht$ cannot be less than or equal to $u.ht$. Contradiction. □

Observation 2: $(\neg u.done \wedge u.ht = p+1) \Rightarrow \neg(\exists v : u \textbf{ nbr } v : v.h \wedge v.ht = p+1)$

Proof: Follows immediately from Observation 1 and Lemma 1. □

Lemma 2: The current point in the computation eventually changes:

$p = k \rightsquigarrow p \neq k$

Proof: By the definition of S, we know that at each point in the trace either (a) exactly one process changes state, or (b) each tuple in the suffix of S is identical (the state of quiescence). We show that in either case, the current point in the computation changes.

Case (a): Some process, say u, changes state in the trace: $p = k \wedge S_{u,k} \neq S_{u,k+1}$

If u is thinking in the current point of the trace (i.e., $S_{u,p} = t$), then, by (S3), $S_{u,p+1} = h$. From (U5), however, the predicate $S_{u,p} = t \wedge S_{u,p+1} = h$ is transient. Since S is constant, the only way for this predicate to be transient is for p to change value. The argument is exactly the same for a process that is thinking in the current point of the trace.

If u is hungry in the current point of the trace (i.e., $S_{u,p} = h$), then, by (S4), $S_{u,p+1} = e$. We show $u.h \wedge u.ht = p+1$:

$$\begin{array}{ll} & S_{u,p} = h \wedge S_{u,p+1} = e \\ \equiv & \quad \{ \text{ def'n } u.done \equiv (\forall i : i \geq p : S_{u,i} \in \{e,t\}) \quad \} \end{array}$$

$$
\begin{array}{ll}
& S_{u,p} = h \ \wedge \ S_{u,p+1} = e \ \wedge \ \neg u.done \\
\Rightarrow & \quad \{ \ \ u.next = (\ \mathbf{Min}\, i \ : \ i \geq p \ \wedge \ S_{u,i} = h \ \wedge \ S_{u,i+1} = e \ : \ i+1 \ ) \ \ \} \\
& S_{u,p} = h \ \wedge \ \neg u.done \ \wedge \ u.next = p+1 \\
\Rightarrow & \quad \{ \ \text{(H2)} \ \neg u.done \ \Rightarrow \ u.ht = u.next \ \ \} \\
& S_{u,p} = h \ \wedge \ u.ht = p+1 \\
\equiv & \quad \{ \ \text{Lemma 1} \ \} \\
& u.h \ \wedge \ u.ht = p+1
\end{array}
$$

By Observation 1, u is the highest priority process among its neighbors v for which $\neg v.done$ holds. Thus, by Lemma 1, u is the highest priority process among its neighbors which are or which could ever again become hungry. This process eventually holds all its forks and $u.mayeat$ holds. Because of u's high priority, it does not relinquish forks until it eats, and therefore $u.mayeat$ continues to hold until action E'_u is executed, and p's value is changed.

Case (b): No process changes state in the trace: $p = k \ \wedge \ S_k = S_{k+1}$.

Action A in $unhygienic'_u$ ensures that the predicate $p = k \ \wedge \ S_p = S_{p+1}$ is transient. Since S is constant, the only way for this predicate to be transient is for p to change value.

By cases (a) and (b), p is guaranteed to eventually change value. □

Lemma 3: The current point in the computation eventually advances by one:

$$p = k \rightsquigarrow p = k+1$$

Proof: Follows from Lemma 2 and the safety property $p = k$ **unless** $p = k+1$ (properties (H1) and (U4)). □

Lemma 4: Any guard added to an existing action in $unhygienic'_u$ can only be falsified by the action it guards.

Proof: The lemma holds trivially for actions H'_u and R'_u. It remains to be shown that $u.done \ \vee \ u.ht = p+1$ is preserved by all actions in $user' \parallel unhygienic'$ except E'_u.

From (H4), $u.done$ is stable, so we must only show that $u.ht = p+1$ is preserved by the execution of any other action when $\neg u.done$ holds. By (H5), $S_{u,p} = h \ \wedge \ S_{u,p+1} = e$. Then action A is not enabled. Furthermore, by Observation 2, no action by any process v for which $\neg v.done$ can execute E'_v and falsify $u.ht = p+1$.

Finally, by Lemma 1, $u.h$ holds and, by (U2), $p = k$ is stable in $user'$. Since $user'$ does not introduce variables which are existing program variables of $unhygienic$, no action of $user'$ can falsify $u.ht = p+1$. □

Lemma 5: Each additional guard in $unhygienic'_u$ is infinitely often true.

Proof: We must show that $u.done \ \vee \ u.ht = p+1$ is infinitely often true.

We know $u.done$ is stable, so if $u.done$ ever holds, the guard will be continuously and therefore infinitely often true.

Now assume, for any point in the computation, $\neg u.done$ holds. Then $u.ht = u.next$ by H4 and by the definition of $u.next$, $u.ht > p$. Also, by (H3), $u.ht$ does not change unless $p = u.ht - 1$. By induction on Lemma 3, eventually $u.ht = p+1$ holds. □

We use the following observation in the proof of Lemma 6.

Observation 3: $(\forall i \ : \ S_{u,i} = e \ : \ (\forall v \ : \ u \ \mathbf{nbr} \ v \ : \ S_{v,i-1} \neq e \ \wedge \ S_{v,i} \neq e \ \wedge \ S_{v,i+1} \neq e \ ) \ )$

Proof: For all neighbors v of u:

$$
\begin{array}{ll}
& S_{u,i} = e \\
\equiv & \quad \{ \ \text{(S6)} \ \} \\
& S_{u,i} = e \ \wedge \ S_{v,i} \neq e \\
\equiv & \quad \{ \ \text{(S0)} \ \} \\
& (S_{u,i-1} = e \ \vee \ S_{v,i-1} \neq e) \ \wedge \ S_{u,i} = e \ \wedge \ S_{v,i} \neq e \ \wedge \ (S_{u,i+1} = e \ \vee \ S_{v,i+1} \neq e) \\
\Rightarrow & \quad \{ \ \text{(S6)} \ \} \\
& S_{v,i-1} \neq e \ \wedge \ S_{v,i} \neq e \ \wedge \ S_{v,i+1} \neq e
\end{array}
$$

□

Lemma 6: The assignment $u.ht := u.next$ in E'_u implements the random assignment in E_u.

Proof: There are two proof obligations: (i) $u.ht$ increases from its previous value, and (ii) $u.ht$ remains distinct from neighbor heights.

Let p_{pre} and $u.ht_{pre}$ denote the value of p and $u.ht$ respectively at the beginning of the execution of E'_u. Consider the value of $u.next$ at the point that the assignment to $u.ht$ occurs, given that the action is enabled:

$u.ht_{pre} = p_{pre} + 1$
$\Rightarrow$ { p incremented before assigning $u.ht$ }
$u.ht_{pre} = p$
$\Rightarrow$ { from def'n of $u.next$, $u.next > p$ }
$u.ht_{pre} < u.next$

Thus, the assignment strictly increases the value of $u.ht$

In order to show (ii), we first show that $S_{u,p+1} = e \;\Rightarrow\; (\forall\, v :\; u$ **nbr** $v :\; v.ht \neq p+2 \;\wedge\; v.ht \neq (\mathbf{Min}\, i :\; i > p \;\wedge\; S_{u,i} = h \;\wedge\; S_{u,i+1} = e :\; i+1\;)\;)$.

In the following, assume v is an arbitrary process such that v **nbr** u.

First assume $v.done \;\wedge\; S_{u,p+1} = e$. If initially $v.done$ then the consequence holds by the **initially** predicate. If initially $\neg v.done$ then at some point in the computation, say $j < p$, v began eating for the last time. Then $v.ht = j + 1 = (\mathbf{Max}\, i :\; S_{v,i-1} = h \;\wedge\; S_{v,i} = e :\; i+1\;)$. It follows $v.ht \neq p+2 \;\wedge\; v.ht \neq (\mathbf{Min}\, i :\; i > p \;\wedge\; S_{u,i} = h \;\wedge\; S_{u,i+1} = e :\; i+1\;)$ by $j < p$ and Observation 3.

Now assume $\neg v.done \;\wedge\; S_{u,p+1} = e$.

$\neg v.done \;\wedge\; S_{u,p+1} = e$
$\Rightarrow$ { (H5) }
$S_{v,v.ht} = e \;\wedge\; S_{u,p+1} = e$
$\Rightarrow$ { Observation 3 }
$v.ht \neq p+2 \;\wedge\; S_{v,v.ht=e}$
$\Rightarrow$ { (S6) }
$v.ht \neq p+2 \;\wedge\; v.ht \neq (\mathbf{Min}\, i :\; i > p \;\wedge\; S_{u,i} = h \;\wedge\; S_{u,i+1} = e :\; i+1\;)$

Then $S_{u,p+1} = e \;\Rightarrow\; (\forall\, v :\; u$ **nbr** $v :\; v.ht \neq p+2 \;\wedge\; v.ht \neq (\mathbf{Min}\, i :\; i > p \;\wedge\; S_{u,i} = h \;\wedge\; S_{u,i+1} = e :\; i+1\;)\;)$. By (H5) $S_{u,p+1} = e$ holds at the beginning of execution of E'_u. Then by sequential composition and the definition of $u.next$, $v.ht \neq u.next$ at the point of the random assignment and heights of neighbors remain distinct. □

Theorem 1: Any fair execution of $unhygienic' \parallel user'$ is a fair execution of $unhygienic \parallel user$.

Proof: Lemmas 4 and 5 establish that each action is executed infinitely often when the additional guard is true. This corresponds to a weakly fair scheduling of actions in the original program. Furthermore, Lemma 6 proves that the deterministic assignment to $u.ht$ in $unhygienic'$ is a possible random assignment to $u.ht$ in $unhygienic$. □

Theorem 2: The $unhygienic$ algorithm, when composed with a $user$ program meeting the requirements described, is a maximal solution to the dining philosophers problem.

Proof: Assume S is an arbitrary trace satisfying the dining philosophers problem. We show that S is a possible execution of $unhygienic \parallel user$.

By Lemmas 1 and 3, S is a possible fair execution of $unhygienic' \parallel user'$. Therefore, by Theorem 1, S is a possible execution of $unhygienic \parallel user$.

Since S was arbitrary, any trace satisfying the specification of the dining philosophers problem is a possible execution of $unhygienic \parallel user$. Therefore the $unhygienic$ algorithm, when composed with a $user$ program meeting the requirements described, is a maximal solution to the problem. □

VI. Discussion

A. Limitations of our Approach

Our approach to proving the maximality of composed systems involves making assumptions about the other programs in the system. While this approach is modular in that it allows us to isolate our attention to a single component, it is not perfectly general. Different restrictions on program behavior are likely required for different systems.

The main limitation of our approach is that the technique is sound, but not necessarily complete. That is, having decided on a set of properties required for the environment in order to prove maximality of the composed system, there may be environments that do not satisfy these properties but still yield maximal systems.

A final limitation of this strategy is that it does not say anything about the maximality of the component in the context of nonmaximal environments. For example, consider a $user$ process that can become hungry at most k

times. It would appear that the unhygienic algorithm can generate all possible traces given the constraint of working with such a $user$. The resulting composed system is not maximal, but the lack of maximality, in a sense, is due entirely to the $user$. It is awkward to show such properties using our method.

B. Maximality and Concurrency

In [5], maximality is associated with concurrency since programs that are maximal admit "maximal concurrency". While this is true, there is no guarantee that a particular execution of a maximal program is "maximally concurrent." Maximally concurrent executions may not even be likely; as true as it is that maximal programs admit maximal concurrency they also admit all other degrees of concurrency.

VII. Conclusion

The noncompositionality of maximality poses a significant challenge to the verification of this property in layered systems. We have developed a method for proving the maximality of composed programs. This method is modular in that it permits these proofs to be carried out in the absence of other component implementations. We have applied this technique in the proof of the first known maximal solution to the diners problem, the unhygienic algorithm.

References

[1] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[2] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[3] Mohamed G. Gouda. *Elements of network protocol design*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[4] Rajeev Joshi and Jayadev Misra. Maximally concurrent programs. Technical Report CS-TR-99-15, 1, 1999.

[5] Rajeev Joshi and Jayadev Misra. Toward a theory of maximally concurrent programs. In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 319–328, New York, NY, USA, 2000. ACM Press.

[6] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[7] Jayadev Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Sprinver-Verlag, New York, NY, USA, 2001.

[8] David Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, 1981. Springer-Verlag.

[9] Scott M. Pike. *Distributed Resource Allocation with Scalable Crash Containment*. PhD thesis, The Ohio State University, Department of Computer Science & Engineering, Aug 2004.

[10] Paolo A.G. Sivilotti, Scott M. Pike, and Nigamanth Sridhar. A new distributed resource-allocation algorithm with optimal failure locality. In *Proceedings of the 12th IASTED Internation Conference on Parallel and Distributed Computing and Systems*, volume 2, pages 524–529. IASTED/ACTA Press, November 2000.