Generalised Dining Philosophers as Feedback Control

Venkatesh Choppella[1], Arjun Sanjeev[1(✉)], Kasturi Viswanath[1], and Bharat Jayaraman[2]

[1] International Institute of Information Technology, Hyderabad, India
arjun.sanjeev@research.iiit.ac.in
[2] University at Buffalo (SUNY), New York, USA

Abstract. We examine the mutual exclusion problem of concurrency through the systematic application of modern feedback control theory, by revisiting the classical problem involving mutual exclusion: the Generalised Dining Philosophers problem. The result is a modular development of the solution using the notions of system and system composition in a formal setting that employs simple equational reasoning. The modular approach separates the solution architecture from the algorithmic minutiae and has the benefit of simplifying the design and correctness proofs.

Two variants of the problem are considered: centralised and distributed topology with N philosophers. In each case, solving the Generalised Dining Philosophers reduces to designing an appropriate feedback controller.

Keywords: Feedback control · Modular concurrency · Generalised Dining Philosophers

1 Introduction

Resource sharing amongst concurrent, distributed processes is at the heart of many computer science problems, specially in operating, distributed embedded systems and networks. Correct sharing of resources amongst processes must not only ensure that a single, non-sharable resource is guaranteed to be available to only one process at a time (safety), but also starvation-freedom – a process waiting for a resource should not have to wait forever. Other complexity metrics of interest (though outside the scope of this paper) are average or worst case waiting time, throughput, etc. Distributed settings introduce other concerns: synchronization, faults, etc.

In this paper, we carry out a systematic application of control systems theory to revisit one of the classical problems in concurrency - the Generalised Dining Philosophers problem.

Mutual exclusion problems are stated in terms of global properties on the combined behaviour of multiple actors running concurrently. One way of solving the mutual exclusion problem is to define a more complex dynamics that

D. V. Hung and M. D'Souza (Eds.): ICDCIT 2020, LNCS 11969, pp. 144–164, 2020.
https://doi.org/10.1007/978-3-030-36987-3_9

each actor (process) implements so as to ensure the required behaviour hold. In the control approach, however, additional actors, *controllers*, are employed that restrain the main actors' original actions in some specific and well-defined way so as to achieve the same. The role of a controller is to issue *commands* to the main actors controlling their behaviour. At the same time, the control exercised on an actor should be not overly constraining: a philosopher should be allowed to exercise his/her choice about what to do next as long as it does not violate the safety conditions.

The objective of this paper is to demonstrate the usefulness of the idea of control, particularly that which involves feedback. Feedback control, also called supervisory control, is the foundation of much of engineering science and is routinely employed in the design of embedded systems. However, its value as a software architectural design principle is only insufficiently captured by the popular "model-view-controller" (MVC) design pattern [1], usually found in the design of user and web interfaces. For example, MVC controllers implement open loop instead of feedback control. Furthermore, they are elements of an architectural design pattern that is not designed to address liveness properties.

A key motivation for the controller based approach is modularity of design. Composition is defined with respect to an interconnect that relates the states and inputs of the two systems being composed [2]. Viewed from this perspective, the Dining Philosophers form systems consisting of interconnected subsystems. A special case of the interconnect which relates inputs and outputs yields modular composition and allows them to be treated as instances of feedback control. The solution then reduces to designing two types of components, the main actors and the controller and their interconnections (the system architecture), followed by definitions of their transition functions.

The compositional approach encourages us to think of the system in a modular way, emphasising the interfaces between components and their interconnections. One benefit of this approach is that it allows us to define multiple types of controllers that interface with a fixed model system. The modularity in architecture also leads to modular correctness proofs of safety and starvation freedom. For example, the proof of the distributed solution to the dining philosophers is reduced to showing that the centralised controller state is reconstructed by the union of the states of the distributed local controllers[1]. That said, however, subtle issues arise even in the simplest variants of the problem. These have to do with non-determinism, timing and feedback, but equally, from trying to seek a precise definition of the problem itself[2].

2 Systems Approach

The main idea in control theory is that of a *system*. Systems have *state* and exhibit behaviour governed by a *dynamics*. A system's state undergoes change

[1] Due to page limit constraints, all proofs are delegated to a technical report [3].

[2] "In the design of reactive systems it is sometimes not clear what is given and what the designer is expected to produce." Chandy and Misra [4, p. 290].

due to input. Dynamics is the unfolding of state over time, governed by laws that relate input with the state. A system's dynamics is thus expressed as a relation between the current state, the current input and the next state. Its output is a function of the state. Thus inputs and outputs are connected via state. The system's state is usually considered hidden, and is inaccessible directly. The observable behaviour of a system is available only via its output. A schematic diagram of a system is shown in Fig. 1.

In control systems, we are given a system, often identified as the *plant*. The plant exhibits a certain *observable behaviour*. In addition to the plant, we are also given a *target behaviour* that is usually a restriction of the plant's behaviour.

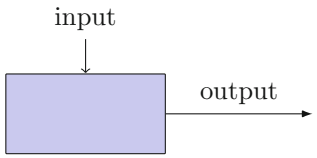

Fig. 1. System with input and output

The control problem is to determine *what additional* input(s) have to be supplied to the plant, such that the resulting dynamics as determined by a new relation between states and inputs now exhibits output behaviour that is as close as possible to the target behaviour specified in the problem. The additional input is usually called the *forced* or *control input*. The plant's dynamics may need to be altered to take into account the combined effect of the original input and the control input.

The second design question is *how* should the control input be computed. Often the control input is computed as a function of the output of the plant (now extended with the control input). Thus we have another system, the controller, one of whose inputs is the output of the plant and whose output is (one of) the inputs to the plant. This architecture is called *feedback control*. The relation between the controller's input and its state and output is called a *control law*. The principle of feedback control is well studied and is used extensively in building large-scale engineering systems of wide variety. Figure 2 is a schematic diagram representing a system with feedback control.

In the rest of the paper, we follow the formal notion of a system and system composition as defined by Tabuada [2]. A complex system is best described as a composition of interconnected subsystems. We employ the key idea of an *interconnect* between two systems, which is a relation that relates the states and the inputs of two systems. While interconnects can, in general, relate states, the interconnects designed in this paper are *modular*: they relate inputs and outputs. Defining a modular interconnect is akin to specifying a wiring diagram between two systems. Modular interconnects drive modular design.

We also illustrate the approach of modular design using feedback control by formulating solutions to the well-known mutual exclusion problem - the generalized dining philosophers, which is an allegorical example of the mutual exclusion problem on an arbitrary graph of processes. This problem is used to introduce the idea of a centralised (hub) controller and also local controllers for the distributed case.

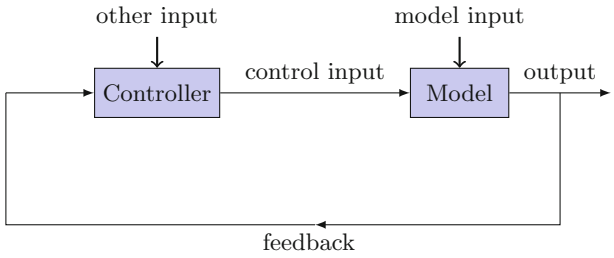

Fig. 2. Feedback control system

3 The Generalised Dining Philosophers Problem

The Dining Philosophers problem, originally formulated by Edsger Dijkstra in 1965 and subsequently published in 1971 [5] is a celebrated thought experiment in concurrency control. A generalization of this problem was suggested and solved by Dijsktra himself [6]. later, the Generalised Dining Philosophers problem was discussed at length by Chandy and Misra's [4]. Each philosopher denotes a process running continuously and forever that is either waiting (*hungry*) for a resource (like a file to write to), or using that resource (*eating*), or having relinquished the resource (*thinking*). A philosopher may either eat or think, but can also get hungry in which case she needs to ensure mutual exclusion: that no adjacent philosopher is eating. The problem consists of designing a protocol by which no philosopher remains hungry indefinitely, the *starvation-freeness* condition, assuming each eating session lasts only for a finite time In addition, *progress* means that there should be no deadlock: at any given time, at least one philosopher that is hungry should move to eating after a bounded period of time. Note that starvation-freedom implies progress.

Individual Philosopher Dynamics: Consider a single philosopher who may be in one of three states: thinking, hungry and eating. At each step, the philosopher may *choose* to either continue be in that state, or switch to the next state (from thinking to hungry, from hungry to eating, or from eating to thinking again). The dynamics of the single philosopher is shown in Fig. 3. Note that the philosophers run forever.

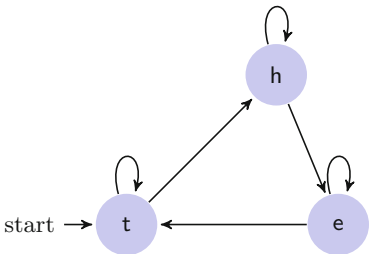

Fig. 3. Philosopher states and transitions

Definition 1 (Generalised Dining Philosophers problem). *N philosophers are arranged in a connected* conflict *graph* $G = \langle V, E \rangle$ *where* V *is a set of* $N = |V|$ *philosophers and* E *is an irreflexive adjacency relation between them.*

If each of the N philosophers was to continue to evolve according to the dynamics in Fig. 3, two philosophers sharing an edge in E could be eating together, violating safety. *The Dining Philosophers problem is the following:*

***Problem**: Assuming that no philosopher eats forever in a single stretch, construct a protocol that ensures*

1. ***Safety:** No two adjacent philosophers eat at the same time.*
2. ***Starvation-freedom:** A philosopher that is hungry eventually gets to eat.*
3. ***Non-intrusiveness:** A philosopher that is thinking or eating continues to behave according to the dynamics of Fig. 3.*

In the subsequent sections, we design solutions to the problem by designing appropriate controller systems and composing them with the philosophers forming feedback loops. The solutions may be broadly classified as either centralised or distributed. The centralised approach assumes a central controller that commands the philosophers on what to do next. The distributed approach assumes no such centralised authority; the philosophers are allowed to communicate to arrive at a consensus on what each can do next. In both the approaches, it is interesting to note that the definition of the philosopher system does not change.

3.1 The Philosopher System

A philosopher can be in 3 states: `t` (thinking), `h` (hungry) or `e` (eating). We denote the set of states of the philosopher by *Act* (also called the *activity* set).

$$Act = \{\mathsf{t}, \mathsf{h}, \mathsf{e}\}$$

The philosopher is designed in such a way that the non-determinism involved in the decision of the philosopher to either stay in the same state or switch to a new one is encoded as a *choice input*, and is captured as a binary input b of type B to the system, where

$$B = \{0, 1\}$$

The resultant system is deterministic with respect to the choice input. On choice $b = 0$ the system stays in the same state; on $b = 1$ it switches to the new state.

The interface of the philosopher is also modified to accommodate the additional control input (or *command*) c of type *Cmd* from the controller, where

$$Cmd = \{\mathsf{pass}, !1, !0\}$$

With command c equal to `pass`, the philosopher follows the choice input b. With the command equal to $!\, b$, the choice input is ignored, and the command prevails in determining the next state of the philosopher according to the value of b: stay if $b = 0$, switch if $b = 1$.

The deterministic and transparent philosopher system S may then be defined as follows:

$$S = \langle a : X = Act, a^0 : X^0 = \{\mathsf{t}\}, (b, c) : U = B \times Cmd, f_S \rangle$$

where $f_S : Act \times B \times Cmd \to Act$ is defined as

$$f_S(a, b, \mathsf{pass}) = f_P(a, b) \tag{1}$$
$$f_S(a, \_, ! \, b) = f_P(a, b) \tag{2}$$

where $f_P : Act \times B \to \ Act$ is defined as

$$f_P(a, 0) = stay(a) \tag{3}$$
$$f_p(a, 1) = switch(a) \tag{4}$$

and $stay : Act \to Act$ and $switch : Act \to Act$ are given by

$$stay(a) = a$$
$$switch(\mathsf{t}) = \mathsf{h}$$
$$switch(\mathsf{h}) = \mathsf{e}$$
$$switch(\mathsf{e}) = \mathsf{t}$$

3.2 N Dining Philosophers with Centralised Control

We now look at the centralised controller solution to the problem. We are given a graph $G = \langle V, E \rangle$, with $|V| = N$ and with each of the N vertices representing a philosopher and E representing an undirected, adjacency relation between vertices. The vertices are identified by integers from 1 to N.

Each of the N philosophers are identical and modeled as instances of the system S described in the previous section. These N vertices are all connected to a single controller (called the hub) which reads the activity status of each of the philosophers and then computes a control input for that philosopher. The control input, along with the choice input to each philosopher computes the next state of that philosopher (Fig. 4).

Notation 31. *Identifiers $j, k, l \in V$ denote vertices.*

An activity map $\overline{a} : V \to A$ *maps vertices to their status, whether hungry, eating or thinking.*

A choice map $\overline{b} : V \to B$ *maps to each vertex a choice value.*

A command map $\overline{c} : V \to Cmd$ *maps to each vertex a command.*

If v is a constant, then $\overline{v}$ denotes a function that maps every vertex to the constant v.

The data structures and notation used in the solution are described below:

1. $G = (V, E)$, the graph of vertices V and their adjacency relation E. G is part of the hub's internal state. G is constant throughout the problem.
   We write $\{j, k\} \in E$, or $E(j, k)$ to denote that there is an undirected edge between j and k in G. We write $E(j)$ to denote the set of all neighbours of j.

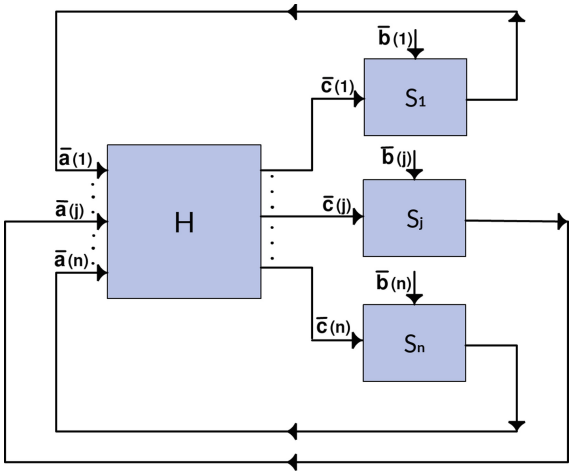

Fig. 4. Wiring diagram describing the architecture of centralised controller.

2. $\overline{a} : V \to \{\mathsf{t}, \mathsf{h}, \mathsf{e}\}$, an activity map. This is input to the hub controller.
3. $D : (j,k) \in E \to \{j,k\}$, is a directed relation derived from E. D is called a *dominance map* or *priority map*. For each edge $\{j,k\}$ of E it returns the source of the edge. The element $\{j,k\} \mapsto j$ of D is indicated $j \mapsto k$ (j dominates k) whereas $\{j,k\} \mapsto k$ is indicated $k \mapsto j$ (j is dominated by k). If $\{j,k\} \in E$, then exactly one of $j \mapsto k \in D$ or $k \mapsto j \in D$ is true.
   $D(j)$ is the set of vertices dominated by j in D and is called the set of *subordinates* of j. $D^{-1}(j)$ denotes the set of vertices that dominate j in D and is called the set of *dominators* of j.
4. $top(D)$, the set of maximal elements of D. $top(D)(j)$ means that $j \in top(D)$. This is a derived internal state of the hub controller.
5. $\overline{c} : V \to Cmd$, the command map. This is part of the internal state of the hub controller and also its output.

Informal Introduction to the Control Algorithm. Initially, at cycle $t = 0$, all vertices in $G = (V, E)$ are thinking, so $\overline{a}[0] = \overline{\mathsf{t}}$. Also, $D[0]$ is D^0, $top(D)[0] = \{j \mid D^0(j) = E(j)\}$ and $\overline{c}[0] = \overline{\mathsf{pass}}$.

Upon reading the activity map, the controller performs the following sequence of computations:

1. (Step 1): Updates D so that (a) a vertex that is eating is dominated by all its neighbours, and (b) any hungry vertex also dominates its thinking neighbours.
2. (Step 2): Computes *top*, the set of top vertices.
3. (Step 3): Computes the new control input for each philosopher vertex: A thinking or eating vertex is allowed to pass. A hungry vertex that is at the top and has no eating neighbours is commanded to switch to eating. Otherwise, the vertex is commanded to stay hungry.

Formal Structure of the Hub Controller. The centralised or hub controller is a deterministic system $H = \langle X, X^0, U, f, Y, h\rangle$, where

1. $X_H = (E \to \mathbb{B}) \times (V \to Cmd)$ is the cross product of the set of all priority maps derived from E with the set of command maps on the vertices of G. Each element $x_H : X_H$ is a tuple $(D, \overline{c})$ consisting of a priority map D and a command map $\overline{c}$.
2. $X_H^0 = (D^0, \overline{c}^0)$ where $D^0(\{j,k\}) = j \mapsto k$ if $j > k$ and $k \mapsto j$ otherwise for $\{j,k\} \in E$, and $\overline{c}^0(j) = \overline{\mathsf{pass}}$. Note that D^0 is acyclic.
3. U_H is the set of activity maps. $\overline{a} : U_H$ represents the activity map that is input to the hub H.
4. $f_H : X, U \to X$ takes a priority map D, a command map $\overline{c}$, and an activity map $\overline{a}$ as input and returns a new priority map D' and a new command map. $f_H((D, \overline{c}), \overline{a}) = (D', g_H(D', \overline{a}))$ where

$$D' = d_H(D, \overline{a}) \tag{5}$$

$$d_H(D, \overline{a}) \stackrel{\mathsf{def}}{=} \{d_H(d, \overline{a}) \mid d \in D\} \tag{6}$$

$$d_H(j \mapsto k, \overline{a}) \stackrel{\mathsf{def}}{=} (k \mapsto j) \quad \text{if } \overline{a}(j) = \mathsf{e} \tag{7}$$

$$\stackrel{\mathsf{def}}{=} (k \mapsto j) \quad \text{if } \overline{a}(j) = \mathsf{t} \text{ and } \overline{a}(k) = \mathsf{h} \tag{8}$$

$$\stackrel{\mathsf{def}}{=} (j \mapsto k) \quad \text{otherwise} \tag{9}$$

Note that the symbol d_H is overloaded to work on a directed edge as well as a priority map. d_H implements the updating of the priority map D to D' mentioned in (Step 1) above. The function g_H computes the command map (Step 3). The command is pass if j is either eating or thinking. If j is hungry, then the command is $!1$if j is ready, i.e., it is hungry, at the top (Step 2), and its neighbours are not eating. Otherwise, the command is $!0$.

$$g_H(D, \overline{a})(j) \stackrel{\mathsf{def}}{=} \mathsf{pass}, \quad \text{if } \overline{a}(j) \in \{\mathsf{t}, \mathsf{e}\} \tag{10}$$

$$\stackrel{\mathsf{def}}{=} !1, \quad \text{if } \mathit{ready}(D, \overline{a})(j) \tag{11}$$

$$\stackrel{\mathsf{def}}{=} !0, \quad \text{otherwise} \tag{12}$$

$$\mathit{ready}(D, \overline{a})(j) \stackrel{\mathsf{def}}{=} \mathsf{true}, \quad \text{if } \overline{a}(j) = \mathsf{h} \wedge j \in \mathit{top}(D) \wedge \forall k \in E(j) : \ \overline{a}(k) \neq \mathsf{e} \tag{13}$$

$$\mathit{top}(D) \stackrel{\mathsf{def}}{=} \{j \in V \| \forall k \in E(j) : j \mapsto k\} \tag{14}$$

5. $Y_H = V \to Cmd$: The output is a command map.
6. $h_H : X_H \to Y_H$ simply projects the command map from its state: $h_H(D, \overline{c}) \stackrel{\mathsf{def}}{=} \overline{c}$.

Note that an existing priority map D when combined with the activity map results in a new priority map D'. The new map D' is then passed to g_H in order to compute the command map.

Wiring the Hub Controller and the Philosophers. Consider the interconnect $\mathcal{I}$ between the hub H and the N philosopher instances s_j, $1 \leq j \leq N$.

$$\mathcal{I} \subseteq X_H \times U_H \times \Pi_{j=1}^{N} s_j.X \times s_j.U$$

that connects the output of each philosopher to the input of the hub, and connects the output of the hub to control input of the corresponding philosopher.

$$\begin{aligned}\mathcal{I} = \{&(x_H, u_H, s_1.x, s_1.u \ldots s_n.x, s_n.u) \mid \\ &u_H(j) = h_S(s_j.x) \ \wedge \ h_H(x_H)(j) = s_j.u, \\ &1 \leq j \leq N\}\end{aligned}$$

The composite N Diners system is the product of the N + 1 systems.

Dynamics of the Centralised N Diners System. The following are the equations that define the dynamics of the centralised N Diners system.

$$\overline{a}^0 = \overline{\mathsf{t}} \tag{15}$$

$$\overline{c}^0 = \overline{\mathsf{pass}} \tag{16}$$

$$D^0 = \{j \mapsto k \mid E(j,k) \wedge j > k\} \tag{17}$$

$$\overline{c} = g_H(D, \overline{a}) \tag{18}$$

$$\overline{a}' = f_S(\overline{a}, \overline{b}, \overline{c}) \tag{19}$$

$$D' = d_H(D, \overline{a}') \tag{20}$$

Asynchronous Interpretation of the Dynamics. It is worth noting that the dynamics of the system as shown above can be interpreted as asynchronous evolution of the philosopher system. A careful examination of the equations yields temporal dependencies between the computations of the variables involved in the systems. Consider the equations, consisting of indexed variables $\overline{a}$, $\overline{c}$ and D:

$$\begin{aligned}\overline{a}[0] &= \overline{\mathsf{t}} \\ D[0] &= D^0 \\ \overline{c}[i] &= g_H(D[i], \overline{a}[i]) \\ \overline{a}[i+1] &= f_S(\overline{a}[i], \overline{b}[i], \overline{c}[i]) \\ D[i+1] &= d_H(D[i], \overline{a}[i+1])\end{aligned}$$

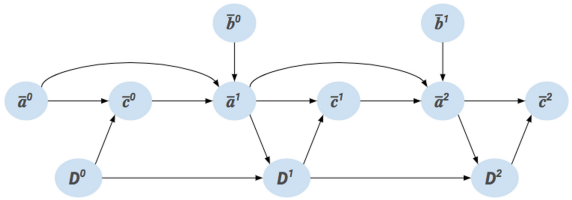

Fig. 5. Dependencies between $\overline{a}$, $\overline{c}$ and D, along with input $\overline{b}$, shown for three calculations.

The asynchronous nature of the system dynamics tells us that the i^{th} value of $\overline{c}$ requires the i^{th} values of $\overline{a}$ and D to be computed before its computation happens, and so on. This implicitly talks about the temporal dependency of the i^{th} value of $\overline{c}$ on the i^{th} values of $\overline{a}$ and D. Similarly, the $(i+1)^{th}$ value of $\overline{a}$ depends on the i^{th} values of $\overline{a}$, $\overline{c}$ and D, and the $(i+1)^{th}$ value of D depends on the i^{th} value of D and the $(i+1)^{th}$ value of $\overline{a}$. Note that they only talk about the temporal dependencies between variable calculations, and do not talk about the clock cycles, nor when the values are computed in physical time. The following figure depicts the dependencies between the variables (Fig. 5).

It is also worth noting that the asynchronous dynamics can be converted to synchronous by composing each system with a global clock and introducing appropriate delays between consecutive computations. This has been discussed in detail in the tech report.

3.3 N Dining Philosophers with Distributed Control

In the distributed version of N Diners, each philosopher continues to be connected to other philosophers adjacent to it according to E, but there is no central hub controller. Usually the problem is stated as trying to devise a protocol amongst the philosophers that ensures that the safety and starvation freedom conditions are met. The notion of devising a protocol is best interpreted as designing a collection of systems and their composition.

Architecture and Key Idea. The centralised architecture employed the global maps $\overline{a}$, $\overline{b}$, $\overline{c}$ and D. While the first three map a vertex j to a value (activity, choice input, or control) the last maps an edge $\{j,k\}$ to one of the vertices j or k.

The key to devising a solution for the distributed case is to start with the graph $G = \langle V, E\rangle$ and consider its distributed representation. The edge relation E is now distributed across the vertex set V. Let α_j denote the size of the set of neighbours $E(j)$ of j. We assume that the neighbourhood $E(j)$ is arbitrarily ordered as a vector $\overrightarrow{E_j}$ indexed from 1 to α_j. Let j and k be distinct vertices in V and let $\{j,k\} \in E$. Furthermore, let the neighbourhoods of j and k be ordered

such that k is the mth neighbour of j and j is the nth neighbour of k. Then, by definition, $\overrightarrow{E_j}(m) = k$ and $\overrightarrow{E_k}(n) = j$.

In addition, with each vertex j is associated a philosopher system S_j and a *local* controller system L_j. The philosopher system S_j is an instance of the system S defined in 3.1. In designing the local controllers, the guiding principle is to distribute the state of the centralised controller to N local controllers. The state of the centralised controller consists of the directed graph D that maps each edge in E to its dominating endpoint and the map $\overline{c} : V \longrightarrow Cmd$ which is also the output of the hub controller.

The information about the direction of an edge $\{j, k\}$ is distributed across two *dominance vectors* $\overrightarrow{d_j}$ and $\overrightarrow{d_k}$. Both are boolean vectors indexed from 1 to α_j and α_k, respectively. Assume that $k = \overrightarrow{E_j}(m)$ and $j = \overrightarrow{E_k}(n)$. Then, the value of $D(\{j, k\})$ is encoded in $\overrightarrow{d_j}$ and $\overrightarrow{d_k}$ as follows: If $D(\{j, k\}) = j$ then $\overrightarrow{d_j}(m) = \mathsf{true}$ and $\overrightarrow{d_k}(n) = \mathsf{false}$. If $D(\{j, k\}) = k$, then $\overrightarrow{d_j}(m) = \mathsf{false}$ and $\overrightarrow{d_k}(n) = \mathsf{true}$.

In the next subsection we define the local controller as a Tabuada system.

Local Controller System for a Vertex j. The controller system L_j has $\alpha_j + 1$ input ports of type A which are indexed 0 to α_j. The output of L_j is of type Cmd.

The local controller L_j is a Tabuada system

$$L_j = \langle X, X^0, U, f, Y, h \rangle$$

where

1. $X = ([1..\alpha_j] \longrightarrow \mathbb{B}) \times Cmd$. Each element of X is a tuple $(\overrightarrow{d_j}, c_j)$ consisting of a *dominance vector* $\overrightarrow{d_j}$ indexed 1 to α_j and a command value c_j. $\overrightarrow{d_j}(m) = \mathsf{true}$ means that the there is a directed edge from j to its mth neighbour k; false means that there is an edge from its mth neighbour to j.
2. X^0 is defined as follows: $X^0 = \langle \overrightarrow{d_j^0}, c_j^0 \rangle$ where $c_j^0 = \mathsf{pass}$ and $\overrightarrow{d_j^0}(m) = \mathsf{true}$ if $\overrightarrow{E_j}(m) = k$ and $j > k$, false otherwise. In other words, there is an edge from j to k if $j > k$.
3. $U = [0..\alpha_j] \longrightarrow A$: We denote the input to L_j as a vector $\overrightarrow{a_j}$, the activities of all the neighbours of the j^{th} philosopher, including its own activity. $\overrightarrow{a_j}(m)$ denotes the value of the mth input port.
4. $f_L : X, U \longrightarrow X$ defines the dynamics of the controller and is given below.
5. $Y = Cmd$, and
6. $h : X \to Y$ and $h(\overrightarrow{d_j}, c_j) = c_j$. The output of the controller L_j is denoted c_j.

The function f_L takes a dominance vector $\overrightarrow{d}$ of length M, a command c and an activity vector $\overrightarrow{a}$ of length $M + 1$ and returns a pair consisting of a new dominance vector $\overrightarrow{d'}$ of length M and a new command c'. f_L first computes the new dominance vector $\overrightarrow{d'}$ using the function d_L. The result $\overrightarrow{d'}$ is then passed along with $\overrightarrow{a}$ to the function g_L, which computes the new command value c'. The functions f_L and d_L are defined below:

$$f_L((\overrightarrow{d}, c), \overrightarrow{a}) = (\overrightarrow{d'}, c') \quad \text{where}$$

$$\overrightarrow{d'} = \overrightarrow{d_L}(\overrightarrow{d}, \overrightarrow{a}), \text{ and} \tag{21}$$

$$c' = g_L(\overrightarrow{d'}, \overrightarrow{a}) \tag{22}$$

$$\overrightarrow{d_L}(\overrightarrow{d}, \overrightarrow{a})(m) \stackrel{\text{def}}{=} d_L(\overrightarrow{d}(m), \overrightarrow{a}(0), \overrightarrow{a}(m)) \quad \text{where } m \in [1..M] \tag{23}$$

$d_L(d, a_0, a)$ is defined as

$$d_L(d, \mathsf{t}, \mathsf{t}) = d \tag{24}$$

$$d_L(d, \mathsf{t}, \mathsf{h}) = \mathsf{false} \tag{25}$$

$$d_L(d, \mathsf{t}, \mathsf{e}) = \mathsf{true} \tag{26}$$

$$d_L(d, \mathsf{h}, \mathsf{e}) = \mathsf{true} \tag{27}$$

$$d_L(d, \mathsf{h}, \mathsf{h}) = d \tag{28}$$

$$d_L(d, \mathsf{e}, \mathsf{h}) = \mathsf{false} \tag{29}$$

$$d_L(d, \mathsf{e}, \mathsf{t}) = \mathsf{false} \tag{30}$$

$$d_L(d, \mathsf{h}, \mathsf{t}) = \mathsf{true} \tag{31}$$

$$d_L(d, \mathsf{e}, \mathsf{e}) = d \tag{32}$$

$d_L(\overrightarrow{d}(m), \overrightarrow{a}(0), \overrightarrow{a}(m))$ takes the mth component of a dominance vector $\overrightarrow{d}$ and computes the new value based on the activity values at the 0th and mth input ports of the controller.

The function g_L takes a dominance vector $\overrightarrow{d}$ of size M and an activity vector $\overrightarrow{a}$ of size $M+1$ and computes a command. It is defined as follows:

$$\begin{aligned} g_L(\overrightarrow{d}, \overrightarrow{a}) &\stackrel{\text{def}}{=} \mathsf{pass}, && \text{if } \overrightarrow{a}(0) \in \{\mathsf{t}, \mathsf{e}\} \\ &\stackrel{\text{def}}{=} !1, && \text{if } \mathit{ready}_L(\overrightarrow{d}, \overrightarrow{a}) = \mathsf{true} \\ &\stackrel{\text{def}}{=} !0, && \text{otherwise} \end{aligned}$$

$$\begin{aligned} \mathit{ready}_L(\overrightarrow{d}, \overrightarrow{a}) &\stackrel{\text{def}}{=} \mathsf{true}, && \text{if } \overrightarrow{a}(0) = \mathsf{h} \text{ and } \mathit{top}_L(\overrightarrow{d}) \text{ and } \forall m \in [1..M] : \overrightarrow{a}(m) \neq \mathsf{e} \\ &\stackrel{\text{def}}{=} \mathsf{false}, && \text{otherwise} \end{aligned}$$

$$\begin{aligned} \mathit{top}_L(\overrightarrow{d}) &\stackrel{\text{def}}{=} \mathsf{true}, && \text{if } \forall m \in [1..M] : \overrightarrow{d}(m) = \mathsf{true} \\ &\stackrel{\text{def}}{=} \mathsf{false}, && \text{otherwise} \end{aligned}$$

Wiring the Local Controllers and the Philosophers. Each philosopher S_j is defined as the instance of the system S defined in Sect. 3.1. Let the choice

input, control input and output of the philosopher system S_j be denoted by the variables $S_j.c$, $S_j.b_\perp$ and $S_j.a$, respectively. The output of L_j is fed as the control input to S_j. The output S_j is fed as 0th input of L_j. In addition, for each vertex j, if k is the mth neighbour of j, i.e., $k = \overrightarrow{E_j}(m)$, then the output of S_k is fed as the mth input to L_j. (See Fig. 6).

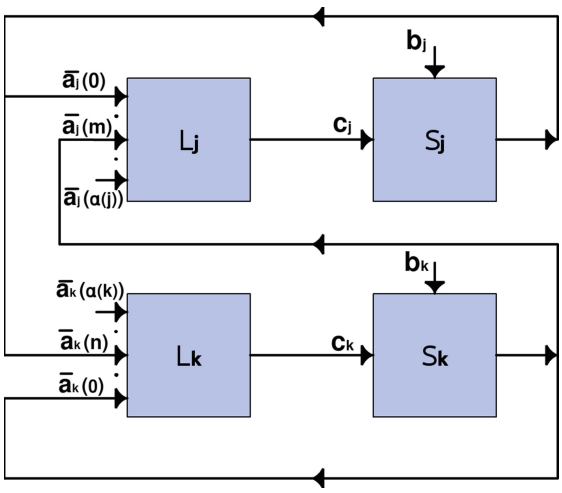

Fig. 6. Wiring between the systems of adjacent philosophers j and k where k and j are respectively the mth and nth neighbour of each other.

The wiring between the N philosopher systems and the N local controllers is the interconnect relation $\mathcal{I} \subseteq \Pi_j S_j.X \times S_j.U \times L_j.X \times L_j.U$, $1 \leq j \leq N$ defined via the following set of constraints:

1. $c_j = S_j.c$: The output of the local controller L_j is equal to the control input of the philosopher system S_j.
2. $S_j.a = \overrightarrow{a_j}(0)$: the output of the philosopher S_j is fed back as the input of the 0th input port of the local controller L_j.
3. $S_k.a = \overrightarrow{a_j}(m)$, where $1 \leq m \leq \alpha_j$ and $k = \overrightarrow{E_j}(m)$: the output of the philosopher S_k is connected as the input of the mth input port of the local controller L_j where k is the mth neighbour of j.
4. $\overrightarrow{d_j}(m) = \neg\overrightarrow{d_k}(n)$, where $k = \overrightarrow{E_j}(m)$ and $j = \overrightarrow{E_k}(n)$. The dominance vector at j is compatible with the dominance vectors of the neighbours of j.

Dynamics of the Distributed N Diners System. Now we can write down the equations that define the asynchronous dynamics of the philosopher system.

Consider any arbitrary philosopher j and its local controller L_j:

$$a_j^0 = \mathsf{t} \tag{33}$$

$$\text{For } m \in [1..\alpha_j]: \quad \overrightarrow{d_j}^0(m) = \mathsf{true}, \quad \text{if } \overrightarrow{E_j}(m) = k \text{ and } j > k$$

$$= \mathsf{false}, \quad \text{otherwise} \tag{34}$$

$$c_j = g_L(\overrightarrow{d_j}, \overrightarrow{a_j}) \tag{35}$$

$$a'_j = f_S(a_j, b_j, c_j) \tag{36}$$

$$\overrightarrow{d_j}' = d_L(\overrightarrow{d_j}, \overrightarrow{a'_j}) \tag{37}$$

From Eq. 36, it can be seen that the philosopher dynamics has not changed - it is the same as that of the centralised case. A close examination of the equations help us deduce that the dynamics we obtained in the distributed case are very much comparable to that of the centralised case. This identical nature of the dynamics form the foundation for the correctness proofs.

Correctness of the Solution to the Distributed Case. The correctness of the solution for the distributed case rests on the claim that under the same input sequence, the controllers and the philosopher outputs in the distributed and centralised cases are identical. This claim in turn depends on the fact that the centralised state may be reconstructed from the distributed state. The detailed proof is given in the technical report [3].

4 Related Work

This section is in two parts: the first is a detailed comparison with Chandy and Misra's solution, the second is a survey of several other approaches.

4.1 Comparison with Chandy and Misra Solution

Chandy and Misra [4] provides the original statement and solution to the Generalised Dining Philosophers problem. There are several important points of comparison with their problem formulation and solution.

The first point of comparison is architecture: in brief, shared variables vs. modular interconnects. Chandy and Misra's formulation of the problem identifies the division between a *user* program, which holds the state of the philosophers, and the *os*, which runs concurrently with the user and modifies variables shared with the *user*. Our formulation is based on formally defining the two main entities, the philosopher and the controller, as formal systems with clearly delineated boundaries and modular interactions between them. The idea of feedback control is explicit in the architecture, not in the shared variable approach.

Another advantage of the modular architecture that our solution affords is apparent when we move from the centralised solution to the distributed solution.

In both cases, the definition of the philosopher remains exactly the same; additional interaction is achieved by wiring a local controller to each philosopher rather than a central controller. We make a reasonable assumption that the output of a philosopher is readable by its neighbours. In Chandy and Misra's solution, the distributed solution relies on three shared boolean state variables per edge in the *user*: a boolean variable *fork* that resides with exactly one of the neighbours, its status *clean* or *dirty*, and a *request token* that resides with exactly one neighbour, adding up to $3|E|$ boolean variables. These variables are not distributed; they reside with the *os*, which still assumes the role of a central controller. In our solution, the distribution of philosopher's *and* their control is evident. Variables are distributed across the vertices: each vertex j with degree j has $\alpha(j)+1$ input ports of type *Act* that read the neighbours' plus self's activity status. In addition, each local controller has, as a boolean vector $\overrightarrow{d}_j$ of length $\alpha(j)$ as part of its internal state, that keeps information about the direction of each edge with j as an endpoint. A pleasant and useful property of this approach is that the centralised data structure D may be reconstructed by the union of local data structures $\overrightarrow{d}$ at each vertex.

The second point of comparison is the algorithm and its impact on reasoning. Both approaches rely on maintaining the dominance graph D as a partial order. As a result, in both approaches, if j is hungry and has priority over k, then j eats before k. In Chandy and Misra's algorithm, however, D is updated only when a hungry vertex transits to eating to ensure that eating vertices are sinks. In our solution, D is updated to satisfy an additional condition that hungry vertices always dominate thinking vertices. This ensures two elegant properties of our algorithm, neither of which are true in Chandy and Misra: (a) a top vertex is also a maximal element of the partial order D, (b) a hungry vertex that is at the top remains so until it is ready, after which it starts eating. In Chandy and Misra's algorithm, a vertex is at the top if it dominates only (all of its) hungry neighbours; it could still be dominated by a thinking neighbour. It is possible that a hungry top vertex is no longer at the top if a neighbouring thinking vertex becomes hungry (Table 1). This leads us to the third property that is true in our approach but not in Chandy and Misra's: amongst two thinking neighbours j and k, whichever gets hungry first gets to eat first.

4.2 Comparison with Other Related Work

Literature on the Dining Philosophers problem is vast. Our very brief survey is slanted towards approaches that—explicitly or implicitly—address the modularity and control aspects of the problem and its solution. [7] surveys the effectiveness of different solutions against various complexity metrics like response time and communication complexity. Here, we leave out complexity theoretic considerations and works that explore probabilistic and many other variants of the problem.

Table 1. Example demonstrating two properties of Chandy and Misra's algorithm: (a) a top hungry vertex no longer remains top, and (b) In step 3, Vertex 1, which was at the top, is hungry, but no longer at the top.

i	G	D	Top	Remarks
0	$\{1:\mathsf{t}, 2:\mathsf{t}, 3:\mathsf{t}\}$	$\{2 \mapsto 1, 3 \mapsto 1\}$	$\{2,3\}$	initial
1	$\{1:\mathsf{h}, 2:\mathsf{t}, 3:\mathsf{h}\}$	ditto	$\{2,3\}$	3 at top
2	$\{1:\mathsf{h}, 2:\mathsf{t}, 3:\mathsf{e}\}$	$\{2 \mapsto 1, 1 \mapsto 3\}$	$\{1,2\}$	1 is at the top
3	$\{1:\mathsf{h}, 2:\mathsf{h}, 3:\mathsf{e}\}$	ditto	$\{2\}$	2 is at the top

Early Works. Dijkstra's Dining Philosophers problem was formulated for the five philosophers seated in a circle. Dijsktra later generalized it to N philosophers. Lynch [8] generalised the problem to a graph consisting of an arbitrary number of philosophers connected via edges depicting resource sharing constraints. Lynch also introduced the notion of an interface description of systems captured via external behaviour, i.e., execution sequences of automata. This idea was popularized by Ramadge and Wonham [9] who advocated that behaviour be specified in terms of language-theoretic properties. They also introduce the idea of control to affect behaviour.

Chandy and Misra [4,10] propose the idea of a dynamic acyclic graph via edge reversals to solve the problem of fair resolution of contention, which ensures progress. This is done by maintaining an ordering on philosophers contending for a resource. The approach's usefulness and generality is demonstrated by their introduction of the Drinking Philosophers problem as a generalisation of the Dining Philosophers problem. In the Drinking Philosophers problem, each philosopher is allowed to possess a subset of a set of resources (drinks) and two adjacent philosophers are allowed to drink at the same time as long as they drink from different bottles. Welch and Lynch [11,12] present a modular approach to the Dining and Drinking Philosopher problems by abstracting the Dining Philosophers system as an I/O automaton. Their paper, however, does not invoke the notion of control. Rhee [13] considers a variety of resource allocation problems, include dining philosophers with modularity and the ability to use arbitrary resource allocation algorithms as subroutines as a means to compare the efficiency of different solutions. In this approach, resource managers are attached to each resource, which is similar in spirit to the local controllers idea.

Other Approaches. Sidhu et al. [14] discuss a distributed solution to a generalised version of the dining philosophers problem. By putting additional constraints and modifying the problem, like the fixed order in which a philosopher can occupy the forks available to him and the fixed number of forks he needs to occupy to start eating, they show that the solution is deadlock free and robust. The deadlock-free condition is assured by showing that the death of any philosopher possessing a few forks does not lead to the failure of the whole network, but instead disables the functioning of only a finite number of philosophers.

In this paper, the philosophers require multiple (more than 2) forks to start eating, and the whole solution is based on forks and their constraints. Also, this paper discusses the additional possibility of the philosophers dying when in possession of a few forks, which is not there in our paper.

Weidman et al. [15] discuss an algorithm for the distributed dynamic resource allocation problem, which is based on the solution to the dining philosophers problem. Their version of the dining philosophers problem is dynamic in nature, in that the philosophers are allowed to add and delete themselves from the group of philosophers who are thinking or eating. They can also add and delete resources from their resource requirements. The state space is modified based on the new actions added: adding/deleting self, or adding/deleting a resource. The main difference from our solution is the extra option available to the philosophers to add/delete themselves from the group of philosophers, as well as add/delete the resources available to them. The state space available to the philosophers is also expanded because of those extra options - there are total 7 states possible now - whereas our solution allows only 3 possible states (thinking, hungry and eating). Also, the notion of a "controller" is absent here - the philosophers' state changes happen depending on the neighbours and the resources availability, but there is no single controller which decides it.

Zhan et al. [16] propose a mathematical model for solving the original version of the dining philosophers problem by modeling the possession of the chopsticks by the philosophers as an adjacency matrix. They talk about the various states of the matrix which can result in a deadlock, and a solution is designed in Java using semaphores which is proven to be deadlock free, and is claimed to be highly efficient in terms of resource usability.

Awerbuch et al. [17] propose a deterministic solution to the dining philosophers problem that is based on the idea of a "distributed queue", which is used to ensure the safety property. The collection of philosophers operate in an asynchronous message-driven environment. They heavily focus on optimizing the "response time" of the system to each job (in other words, the philosopher) to make it polynomial in nature. In our solution, we do not talk about the response time and instead we focus on the modularity of the solution, which is not considered in this solution.

A distributed algorithm for the dining philosophers algorithm has been implemented by Haiyan [18] in Agda, a proof checker based on Martin-Lof's type theory. A precedence graph is maintained in this solution where directed edges represent precedences between pairs of potentially conflicting philosophers, which is the same idea as the priority graph we have in our solution. But unlike our solution, they also have chopsticks modelled as part of the solution in Agda.

Hoover et al. [19] describe a fully distributed self-stabilizing[3] solution to the dining philosophers problem. An interleaved semantics is assumed where only one philosopher at a time changes its state, like the asynchronous dynamics in our solution. They use a token based system, where tokens keeps circling the

[3] Regardless of the initial state, the algorithm eventually converges to a legal state, and will therefore remain only in legal states.

ring of philosophers, and the possession of a token enables the philosopher to eat. The algorithm begins with a potentially illegal state with multiple tokens, and later converges to a legal state with just one token. Our solution do not have this self-stabilization property, as we do not have any "illegal" state in our system at any point of time.

The dining philosophers solution mentioned in the work by Keane et al. [20] uses a generic graph model like the generalized problem: edges between processes which can conflict in critical section access. Modification of arrows between the nodes happens during entry and exit from the critical section. They do not focus on aspects like modularity or equational reasoning, but on solving a new synchronization problem (called GRASP).

Cargill [21] proposes a solution which is distributed in the sense that synchronization and communication is limited to the immediate neighbourhood of each philosopher without a central mechanism, and is robust in the sense that the failure of a philosopher only affects its immediate neighbourhood. Unlike our solution, forks are modelled as part of their solution.

You et al. [22] solve the Distributed Dining Philosophers problem, which is the same as the Generalized Dining Philosophers problem, using category theory. The phases of philosophers, priority of philosophers, state-transitions etc. are modelled as different categories and semantics of the problem are explained. They also make use the graph representation of the priorities we have used in our paper.

Nesterenko et al. [23] present a solution to the dining philosophers problem that tolerates malicious crashes, where the failed process behaves arbitrarily and ceases all operations. They talk about the use of stabilization - which allows the program to recover from an arbitrary state - and crash failure locality - which ensures that the crash of a process affects only a finite other processes - in the optimality of their solution.

Chang [24] in his solution tries to decentralise Dijkstra's solution to the dining philosophers problem by making use of message passing and access tokens in a distributed system. The solution does not use any global variables, and there is no notion of "controllers" in the solution like we have in ours. Forks are made use of in the solution.

Datta et al. [25] considers the mobile philosophers problem in which a dynamic network exists where both philosophers and resources can join/leave the network at any time, and the philosophers can connect/disconnect to/from any point in the network. The philosopher is allowed to move around a ring of resources, making requests to the resources in the process. The solution they propose is self-stabilizing and asynchronous.

Supervisory Control. The idea of using feedback (or supervisory) control to solve the Dining Philosophers program is not new. Miremadi et al. [26] demonstrate how to automatically synthesise a supervisory controller using Binary Decision Diagrams. Their paper uses Hoare composition but does not describe the synthesised controller, nor do they attempt to prove why their solution is

correct. Andova et al. [27] use the idea of a central controller delegating part of its control to local controllers to solve the problem of self-stabilization: i.e., migrating a deadlock-prone configuration to one that is deadlock-free using distributed adaptation.

Similar to our solution, Vaughan [28] presents centralised and distributed solutions to the dining philosophers problem. The centralised solution does not have a hub controller, but has monitor data structures, which store information like the number of chopsticks available to each philosopher, the claims made by a philosopher on his adjacent chopsticks, etc. In his distributed solution, the chopsticks are viewed as static resources and there are manager processes, like we have controllers, to control them. But unlike our solution, the local manager processes only control the chopsticks (with the help of a distributed queue to sequentialize access to the chopsticks for the philosophers) and not the philosophers, and the access to the resources is scheduled by the philosophers by passing messages between themselves.

Siahaan [29], in his solution, proposes a framework containing an active object called "Table" which controls the forks and the state transitions of the philosophers. The other active objects in the framework are the philosophers and the timer controller (which issues timeout instructions to the philosophers to change state). The table manages the state-change requests of the philosophers depending on the state of forks, hence serving a purpose similar to the controllers in our solution. The timer object sends instructions to the philosophers for state change, but our paper does not involve a timer to do so.

Feedback control has been used to solve other problems too. Wang et al. [30] model discrete event systems using Petri nets and synthesise feedback controllers for them to avoid deadlocks in concurrent software. Mizoguchi et al. [31] design a feedback controller of a cyber-physical system by composing several abstract systems, and prove that the controlled system exhibits the desired behaviour. Fu et al. [32] model adaptive control for finite-state transition systems using elements from grammatical inference and game theory, to produce controllers that guarantee that a system satisfies its specifications.

Synchronous Languages. Synchronous languages like Esterel, SIGNAL and Lustre [33] are popular in the embedded systems domain because synchronicity allows simpler reasoning with time. Gamatie [34] discusses the N Dining Philosophers problem with the philosophers seated in a ring. The example is presented in the programming language SIGNAL, whose execution model uses synchronous message passing. The SIGNAL programming language also compiles the specifications to C code. The solution uses three sets of processes: one for the philosophers, one for the forks, and one for the main process used for coordination. Communication between the philosophers and the forks happens via signals that are clocked. In this respect, the solution is similar to the one described in this paper. However, in the solution, each signal has its own clock (polysynchrony), all derived from a single master clock.

5 Conclusion and Future Work

This work has three objectives: first, to apply the idea of feedback control to problems of concurrency; second, to systematically apply the notion of Tabuada systems and composition when constructing the problem statement and its solution, and third, to ensure that the solution is as modular as possible. In the process, we have also come up with a different solution in the case of the Generalised Dining Philosophers problem, one which reveals how the distributed solution is a distribution of the state in the centralised solution.

The solutions discussed in this paper using this approach leads us to believe that this is a promising direction to explore in the future, the formalisation of software architectures for other sequential and concurrent systems.

References

1. Gamma, E., Helm, R., Johnson, R., Visslides, R.: Design Patterns: Elements of Reusable Object-Oriented Software. Addisson-Wesley, Reading (1994)
2. Tabuada, P.: Verification and Control of Hybrid Systems: A Symbolic Approach. Springer, Boston (2009). https://doi.org/10.1007/978-1-4419-0224-5
3. Choppella, V., Viswanath, K., Sanjeev, A.: Generalised dining philosophers as feedback control. arXiv preprint arXiv:1805.02010 (2018)
4. Chandy, M., Misra, J.: Parallel Program Design: A Foundation. Addison–Wesley, Reading (1988)
5. Dijkstra, E.W.: Hierarchical ordering of sequential processes. Acta Informatica **1**, 115–138 (1971). Also published as EWD 310
6. Dijkstra, E.W.: Two starvation-free solutions of a general exclusion problem. Circulated privately (1977)
7. Papatriantafilou, M.: On distributed resource handling: dining, drinking and mobile philosophers. In: Proceedings of the First International Conference on Principles of Distributed Systems (OPODIS), pp. 293–308(1997)
8. Lynch, N.: Upper bounds for static resource allocation in a distributed system. J. Comput. Syst. Sci. **23**, 254–278 (1981)
9. Ramadge, P., Wonham, W.: The control of discrete event systems. Proc. IEEE **77**(1), 81–98 (1989)
10. Chandy, K.M., Misra, J.: The drinking philosophers problem. ACM Trans. Program. Lang. Syst. **6**(4), 632–646 (1984)
11. Welch, J.L., Lynch, N.A.: A modular drinking philosophers algorithm. Distrib. Comput. **6**(4), 233–244 (1993)
12. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, Burlington (1996)
13. Rhee, I.: A fast distributed modular algorithm for resource allocation. In: Proceedings of the 15th International Conference on Distributed Computing Systems, 1995, pp. 161–168, May 1995
14. Sidhu, D.P., Pollack, R.H.: A robust distributed solution to the generalized dining philosophers problem. In: 1984 IEEE First International Conference on Data Engineering, pp. 483–489. IEEE (1984)
15. Weidman, E.B., Page, I.P., Pervin, W.J.: Explicit dynamic exclusion algorithm
16. Zhan, J., Guo, Y., Liu, C.: A deadlock prevention using adjacency matrix on dining philosophers problem, vol. 121–126 (2012)

17. Awerbuch, B., Saks, M.: A dining philosophers algorithm with polynomial response time. In: Proceedings of the 31st Symposium on Foundations of Computer Science (FOCS), pp. 65–74 (1990)
18. Haiyan, Q.: A Distributed Algorithm in Agda: The Dining Philosophers (1999)
19. Hoover, D., Poole, J.: A distributed self-stabilizing solution to the dining philosophers problem. Inf. Process. Lett. **41**(4), 209–213 (1992)
20. Keane, P., Moir, M.: A general resource allocation synchronization problem. In: 21st International Conference on Distributed Computing Systems, 2001, pp. 557–564. IEEE (2001)
21. Cargill, T.A.: A robust distributed solution to the dining philosophers problem. Softw. Pract. Exp. **12**(10), 965–969 (1982)
22. You, Z., Xue, J., Ying, S.: Categorial semantics of a solution to distributed dining philosophers problem. In: Lee, D.-T., Chen, D.Z., Ying, S. (eds.) FAW 2010. LNCS, vol. 6213, pp. 172–184. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14553-7_18
23. Nesterenko, M., Arora, A.: Dining philosophers that tolerate malicious crashes. In: Proceedings on the 22nd International Conference on Distributed Computing Systems, 2002, pp. 191–198. IEEE (2002)
24. Chang, E.: n-philosophers: an exercise in distributed control. Comput. Netw. (1976) **4**(2), 71–76 (1980)
25. Datta, A.K., Gradinariu, M., Raynal, M.: Stabilizing mobile philosophers. Inf. Process. Lett. **95**(1), 299–306 (2005)
26. Miremadi, S., Akesson, K., Fabian, M., Vahidi, A., Lennartson, B.: Solving two supervisory control benchmark problems in Supremica. In: Proceedings of the 9th International Workshop on Discrete Event Systems, pp. 131–136. IEEE (2008)
27. Andova, S., Groenewegen, L.P.J., de Vink, E.P.: Distributed adaption of dining philosophers. In: Barbosa, L.S., Lumpe, M. (eds.) FACS 2010. LNCS, vol. 6921, pp. 125–144. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27269-1_8
28. Vaughan, J.G.: The dining philosophers problem and its decentralisation. Microprocess. Microprogr. **35**(1–5), 455–462 (1992)
29. Siahaan, A.P.U.: Synchronization in dining philosophers problem using lock & release algorithm
30. Wang, Y., Kelly, T., Kudlur, M., Mahlke, S., Lafortune, S.: The application of supervisory control to deadlock avoidance in concurrent software. In: 9th International Workshop on Discrete Event Systems, WODES 2008, pp. 287–292. IEEE (2008)
31. Mizoguchi, M., Ushio, T.: Output feedback controller design with symbolic observers for cyber-physical systems. arXiv preprint arXiv:1612.04974 (2016)
32. Fu, J., Tanner, H.G., Heinz, J., Chandlee, J.: Adaptive symbolic control for finite-state transition systems with grammatical inference. IEEE Trans. Autom. Control. **59**(2), 505–511 (2014)
33. Halbwachs, N.: Synchronous Programming of Reactive Systems. Kluwer Academic Publishers, Dordrecht (1993)
34. Gamatié, A.: Designing Embedded Systems with the Signal Programming Language: Synchronous, Reactive Specification. Springer, New York (2009). https://doi.org/10.1007/978-1-4419-0941-1