REASONING ABOUT DISTRIBUTED SYSTEM SPECIFICATION: THE TEMPORAL-CAUSAL WAY

Jaelson Freire Brelaz de Castro
Departamento de Informática - UFPE
Caixa Postal 7851- 50732 Recife- PE
Brazil
jbc@di.ufpe.br

Abstract

The temporal-causal framework supports the specification of systems from three points of view: informal, causal and temporal-causal. The Informal View *describes in natural language and graphically the requirements of the component. It is based on the* Conic Environment *that provides a language-based approach to the building of distributed system. The* Causal View *relies on enhanced Petri Nets to describe internal structure, distributed control and safety requirements. The third view makes use of temporal-causal logic to describe temporal features of the components such as liveness requirements. Previous causal properties are preserved.*

In this paper we use the technique to specify and reason about properties of distributed systems. Local properties of the components as well as global properties of the systems are derived. The Hygienic Dining Philosopher example is used to the illustrate the viability of the specification languages and adequacy of the logic formalism.

1 Introduction

In [4] we developed a framework to support specification, i.e. modeling and analysis, of concurrent and distributed systems. A *system* is composed of separate, interacting *components*, possibly highly independent of each other. Components will have well-defined interfaces and their local state that changes over time. A system is described and managed in terms of their configuration [6].

We argued that a single representation scheme is often not enough to capture the various features of system behaviour. Instead, multiple viewpoints should be used to partition the domain of information. Its success rests upon the selection of appropriate representation schemes, the careful definition of the relations between them and the process by which such specifications are built within those representation schemes.

Indeed, the major feature of research in software engineering and formal methods in recent years has been the trend towards combinations and integrations of different approaches [8, 9, 10, 3, 2].

We showed how multiple representation schemes, such as temporal logics and Petri Nets, can be used to describe the behaviour of systems. This temporal-causal framework will contribute towards the provision of a more effective basis for software development, enabling system specification from multiple points of view. The main advantages are the ability to express explicitly, clearly and compactly the causal and temporal relationship between the events of the system. Formal reasoning is also supported, i.e local and global properties are derived.

Systems are specified from three points of view: informal, causal and temporal-causal. The *Informal View* describes in natural language and graphically the requirements of the component. It is based on the *Conic Environment* that provides a language-based approach to the building of distributed system. The *Causal View* relies on enhanced Petri Nets to describe internal structure, distributed control and safety requirements. The third view makes use of temporal-causal logic to describe temporal features of the components such as liveness requirements. Previous causal properties are preserved.

In [4] we discuss how Petri Nets and Temporal Logic can be integrated. In essence, the basic Petri Net model is enhanced (to include types, guarded transitions and the annotation of places with a logical scope) and given a logical proof-theoretic characterization. This logic is then merged with standard temporal logic. The resulting logic is called *Temporal-Causal Logic*. A methodology for the temporal-causal specification of systems is briefly discussed in [1]. It shows how one can start from an informal description and end up with a temporal-causal specification.

In this paper we use the technique to specify and reason about properties of distributed systems. Local properties of the components as well as global properties of the systems are derived. The Hygienic Dining Philosopher example is used to the illustrate the viability of the specification languages and adequacy of the logic formalism.

Section 2 introduces the Hygienic Dining Philosopher problem. It reviews the solution proposed by [5]. In **Section 3** we use the temporal-causal framework to specify the basic component of the Hygienic Dining Philosophers System. We then carry on and prove some local properties of the philosopher component.

In **Section 4** we describe a "Diner System" comprised of three philosopher components named "*plato*", "*karl*" and "*max*" which are sitting around a table, with *plato* next to *karl*, who is next to *max*, who in turn is next to *plato*. We then proceed to prove that every hungry philosopher will eat. **Section 5** summarizes the discussions and concludes the paper.

2 Hygienic Dining Philosophers System

Our example is based on the well known *Dining Philosophers Problem*. It has been extensively used in the literature to illustrate the possibility of conflicts between processes in distributed systems. Philosophers are arranged in a ring with neighbouring philosophers sharing a fork. Figure 1 depicts a table with three philosophers. A philosopher is either thinking, hungry or eating. To move from the hungry eating state a philosopher must acquire both his lefthand and righthand fork.

In the sequel we shall describe a solution to the fully distributed diners problem. It is based on the work of Chandy and Misra [5] which relies on two principles:

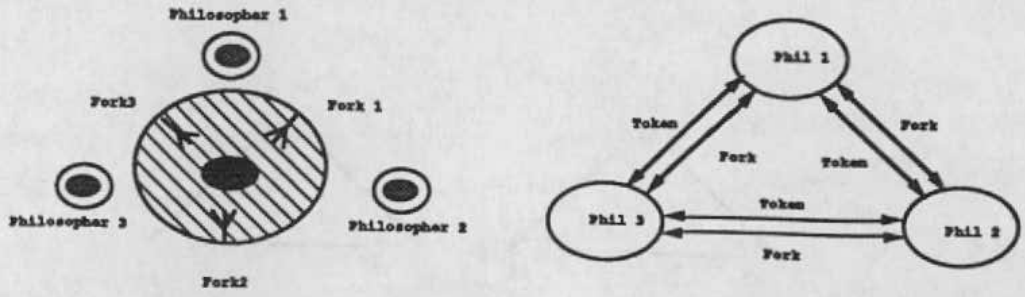


Figure 1: Dining Philosophers Problem

- In every state of the diners system at least one philosopher in every set of conflicting philosophers must be distinguished from the other philosophers of the set;
- The distinguishing property must be such that the philosopher selected for favourable treatment is not always the same, i.e conflicts should not always be resolved to the detriment (or favour) of a particular philosopher.

The former is achieved through a distributed implementation of an acyclic precedence graph, the depth of a philosopher (the longest chain of predecessors) being the distinguishing property. The latter, i.e. fairness of the solution, is obtained through a simple conflict resolution rule coupled with the acyclic graph. The solution of Chandy and Misra can be described informally as: "*A fork is either clean or dirty. A fork being used to eat with is dirty and remains dirty until it is cleaned. A clean fork remains clean until it is used for eating. A philosopher cleans a fork when mailing it (he is hygienic). An eating philosopher does not satisfy requests for forks until he has finished eating*". Moreover, a noneating philosopher defers requests for forks that are clean and satisfies requests for forks that are dirty.

As stated in [7], their solution can be considered to implement a precedence graph such that an edge directed from a node **u** to **v** indicates that **u** has precedence over **v** (Figure 2). In the diners solution a philosopher node **u** has precedence over its neighbour **v** if and only if :

- **u** holds the fork and it is clean;
- **v** holds the fork and it is dirty; and
- the fork is in transit from **v** to **u**.

Furthermore, the direction (from **u** to **v**) of the edge can change only when **u** starts eating and all edges incident on an eating philosopher are directed toward it. Therefore the graphs are acyclic.

The initial conditions are the following:

- all forks are dirty
- forks are distributed among philosophers such that the precedence graph is acyclic
- if **u** and **v** are neighbours then either **u** holds the fork and **v** the priority request token or vice versa..

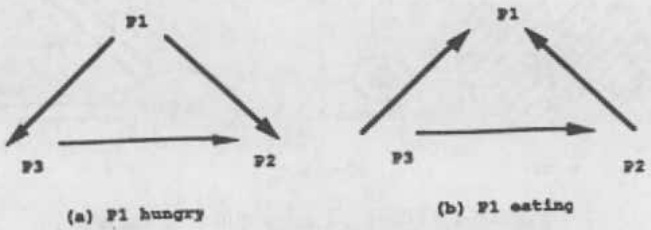


Figure 2: Dining Philosophers: Precedence Graph

We shall use the *Temporal-Causal Framework* described in [4, 2] to specify this solution to the diners problem. The description will consist of three complementary views: Informal, Causal and Temporal-Causal. This specific section concentrates on the specification and analysis of local properties of an *individual philosopher*. We shall delay the description of the full *system*, which is the composition of n individual philosophers, and reasoning of global properties to **Section 4**.

3 Philosopher Component: Temporal-Causal Specification and Analysis

In this section we use the temporal-causal framework (see [4, 3, 1]) to specify the basic component of the Hygienic Dining Philosophers System and prove some local properties of its behaviour.

The *Informal View* describes in natural language and graphically the requirements of the component. It is based on the *Conic Environment* which provides a language-based approach to the building of distributed system. The *Causal View* relies on enhanced Petri Nets to describe internal structure, distributed control and safety requirements. The third view makes use of temporal-causal logic to describe temporal features of the components such as liveness requirements. Previous causal properties are preserved.

3.1 Philosopher's Informal View

The objective of this initial description is to identify the basic features of the philosopher being described. It should make use of both natural language and graphics. Because a system will be composed of separate, interacting philosophers, we shall put emphasis on the identification of messages and ports.

This particular solution relies on the concept of exchange of tokens and forks between neighbouring philosophers. Assuming that fork "f" can take the value of left (l) or right (r), there are the following types of messages:

- **fork**$_f$ passes (clean) fork f to neighbour which shares f;
- **token**$_f$ passes request token for fork f to neighbour which shares f.

Hence, we have the following message types for the dining philosophers system:

```
define philomsg: fork, nametype, phil, position, signaltype, token

type fork       = (left, right);
     nametype   = packed array [1 ... namelength] of char;
     phil       = nametype;
     position   = (left, right);
     signaltype = boolean;
     token      = (left, right);
```

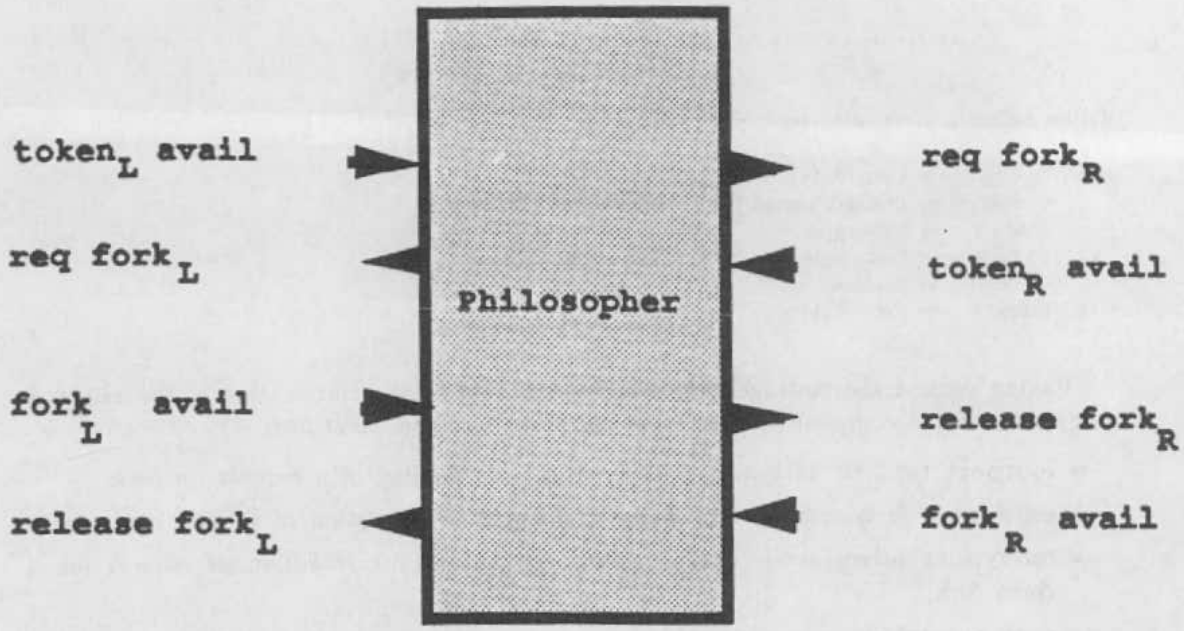
Having defined the messages, the next task should be the clarification of the exit and entryports of the component. The following ports could be identified:

- **exitport** *req fork$_f$* of type *token$_f$* to model the sending of a request for forks;
- **entryport** *fork$_f$ avail* of type *fork$_f$* to indicate the reception of a clean fork ;
- **entryport** *token$_f$ avail* of type *token$_f$* to capture the reception of request for a clean fork;
- **extiport** *release fork$_f$* of type *fork$_f$* to express that a clean fork has been released.

Given this set of messages and ports one can come up with the *Informal View* of Figure 3.

The following rules are expected to informally describe an individual philosopher:

- **R1 Requesting a fork f** if a philosopher holds the token, is hungry but does **not** possess the fork f, he should send the token to his neighbour requesting the fork. The predicate which indicates the possession of the token must be reset (hastoken(f)=false).
- **R2 Releasing a fork f** if a philosopher possess a fork and is **not** eating and provided that the fork is dirty and he possesses the request token, the philosopher should send a message to his neighbour releasing the fork . Forks are cleaned before they are sent (dirty(f)=false and hasfork(f)=false).
- **R3 Receiving a request token for f** a philosopher shall be able to receive a request for token, in which case a predicate indicating the possession of the token must be set (hastoken(f)=true).
- **R4 Receiving a fork f** a philosopher shall be able to receive a fork, in which case a predicate indicating the possession of a fork should be set (hasfork(f)=true). Because the forks are clean then predicate which indicates the dirtiness of a fork should be unset (dirty(f)=false).
- **R5 Philosopher Starts Thinking** from a "eating condition" a philosopher should move to a "thinking condition" provided that predicate "Eating timeout" holds.
- **R6 Philosopher Becomes Hungry** from a "thinking condition" a philosopher should move to a "hungry condition" provided that predicate "Thinking timeout" holds.
- **R7 Philosopher Starts eating** from a "hungry condition" a philosopher may move to a "eating condition" provided that he possesses both both his left and right forks. Note that before the transition it should not matter if the forks are clean or dirty. However after the transition both forks are dirtied (dirty(l)=true and dirty(r)=true).



A philosopher is either thinking,hungry or eating. To move from hungry to the eating state a philosopher must acquire both his lefthand and righthand fork (**receive** fork$_f$ **from** fork$_f$ avail). If he does not yet possess them, a request for fork should be issued (**send** token$_f$ **to** req fork$_f$). A fork being used to eat with is dirty and remains dirty until it is cleaned. A clean fork remains clean until it is used for eating. A philosopher may receive requests for forks (**receive** token$_f$ **from** token$_f$ avail). However, an eating philosopher does not satisfy them until he has finished eating. When not eating, philosophers defers requests for forks that are clean and satisfy requests for forks that are dirty (**send** fork$_f$ **to** release fork$_f$).

Figure 3: Hygienic Philosopher Component: Informal View

3.2 Philosopher's Causal View

Before we get on to the discussion of the philosopher causal description (see Figure 4), it is perhaps appropriate to remind the reader of some conventions. At the highest level of abstraction, a component can be viewed as a transition with input (entry) and output (exit) places. Those interface places must correspond to entryport and exitports of the informal view. Places are typed according to the message that it is supposedly conveying. Graphically, interface places are differentiated from the other normal places (transparent circle) by a different filling pattern (see Figure 4).

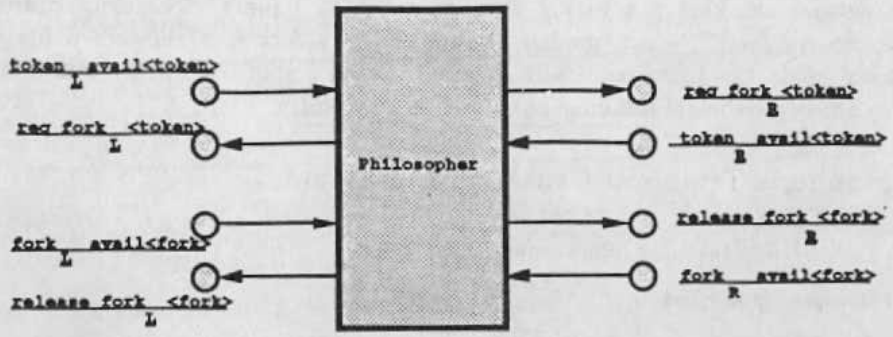


Figure 4: Hygienic Philosopher Component: Causal View

We shall adopt the convention of underlining condition predicates which correspond to places. Typical examples of "entry" places are $\underline{token_f\ avail\langle token\rangle}$ and $\underline{fork_f\ avail\langle fork\rangle}$. For "exit" places we have $\underline{req\ fork_f\langle token\rangle}$ and $\underline{release\ fork_f\langle fork\rangle}$.

Places may have variable extensions. That is, they may represent changing properties of individuals. For instance, place $\underline{Holding\ Fork\langle fork\rangle}$ may have tokens denoting the presence of the left fork ($fork_l$) or the right fork ($fork_r$). A label underneath the arc will indicate which objects are affected by the transition. For example when transition "**receive** $fork_l$ **from** $fork_l$ *avail*" fires, the label "$fork_l$" underneath the arc indicates that the condition predicate $\underline{Holding\ Fork\langle fork\rangle}$ holds for $fork_l$ in that slice. Graphically it is as if the circle associated with this condition predicate were able to hold two distinct dots ($fork_l$ and/or $fork_r$).

As far as transitions are concerned, it should not be difficult to identify them from the set of rules (R1-R7) presented in the informal view. Observe that each interface place has associated with it a communication transition or event. Typical examples are: "**receive** $token_f$ **from** $token_f$ *avail*", "**send** $token_f$ **to** *req* $fork_f$", "**receive** $fork_f$ **from** $fork_f$ *avail*" and "**send** $fork_f$ **to** *release* $fork_f$". Three remaining transitions have to do with the basic cycle of activities of a philosopher: start thinking, become hungry, start eating.

Note that there are two types of transitions: normal (solid bars) and conditional (hatched bars). Conditional transitions (events), as the name suggests, requires the holding of the "guard" for it to fire (occur). It is natural to expect that the "provisos" included in the informal rules (**R2, R5, R6** and **R7**) act as guards. For instance, from **R5** we shall note that a thinking philosopher can become hungry, *provided* that the proper thinking timeout has expired.

A transition may have a {scope} associated with it. The scope is a well-formed-formula of the enhanced net language which holds after the firing (occurrence) of the transition

(event). For example the reception of a token (transition **receive** *token$_f$* **from** *token$_f$ avail*) causes the predicate hastoken(f) to hold.

Note the presence of inhibitor arcs. Graphically they are represented as dotted lines which have a black circle at their end. Often, inhibitor arcs are used to link input places to transitions. The presence of a token in one such place will disallow the firing of the transition. For instance, consider transition "**send** *fork$_f$* **to** release *fork$_f$*", which models the release of forks. From the diagram one can note that the philosopher defers request for forks that are clean and when **not** eating satisfy requests for forks that are dirty.

Moreover, since a philosopher is always either eating, thinking or hungry, it is fair to say if he is neither thinking nor eating then he must be hungry. Therefore, transition "**send** *token$_f$* **to** *req fork$_f$*", which models the request of a fork f, is allowed to fire when the philosopher holds the token and is in a hungry state (modelled by the two inhibitor arcs from condition predicate *thinking⟨phil⟩* and *eating⟨phil⟩*).

3.3 Philosopher's Temporal-Causal Specification

Philosopher Specification

Component Philosopher (*fork$_l$*?, *fork$_r$*?);

import philomsg: token, fork, position, phil, signaltype;
exitport *req fork$_f$*: token;
entryport *token$_f$ avail*: token;
exitport *release fork$_f$*: fork;
entryport *fork$_f$ avail*: fork;

New type philosopher;

Variables
p : ⟨phil⟩;
signal : ⟨signaltype⟩;
fork$_f$: ⟨fork⟩;
token$_f$: ⟨token⟩;
f : ⟨position⟩;

Condition Predicates
token$_f$ avail, *reqf*, *req fork$_f$*: ⟨token⟩;
Thinking, *Hungry*, *Eating*: ⟨phil⟩;
fork$_f$ avail, *Holding Fork*, *release fork$_f$*: ⟨fork⟩;

Predicates
hastoken: ⟨token⟩;
dirty, hasfork: ⟨fork⟩;
Eating timeout: ⟨signaltype⟩;
Thinking timeout: ⟨signaltype⟩;

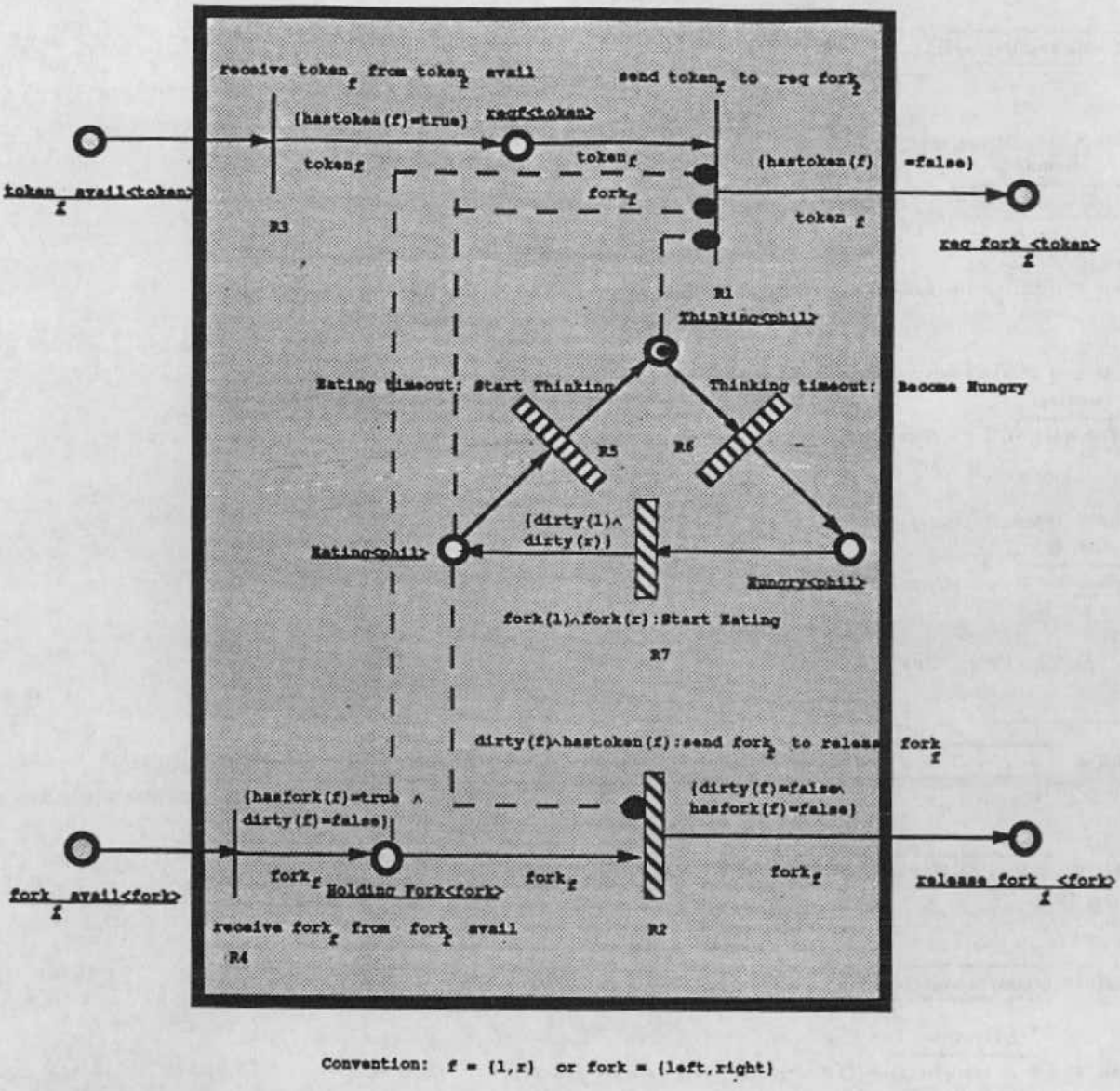
Events

send *token$_f$* **to** *req fork$_f$*: ⟨ token⟩;	(to be used with **R1**)
send *fork$_f$* **to** *release fork$_f$*: ⟨fork⟩;	(to be used with **R2**)
receive *token$_f$* **from** *token$_f$ avail*: ⟨token⟩;	(to be used with **R3**)
receive *fork$_f$* **from** *fork$_f$ avail*: ⟨fork⟩;	(to be used with **R4**)
Start Thinking: ⟨phil⟩;	(to be used with **R5**)
Become Hungry: ⟨phil⟩;	(to be used with **R6**)
Start Eating: ⟨phil⟩;	(to be used with **R7**)

Behaviour

Causal View (Figure 5);

Temporal-Causal View:



Convention: f = (l,r) or fork = (left,right)

Figure 5: Hygienic Philosopher: Enhanced Net Description

Causal Properties

phil 1 ([*send token$_f$ to req fork$_f$*) $\overbrace{\underline{req\ fork_f(token_f)}}^{post-condition} \wedge \overbrace{\{\neg hastoken_f(f)\}}^{scope} \wedge \overbrace{\underline{\neg reqf(f)}}^{resetting}$; *(from* **[R1]***)*

phil 2 ([*dirty(f)*$\wedge$*hastoken(f) : send fork$_f$ to release fork$_f$*) $\overbrace{\underline{release\ fork_f(f)}}^{post-condition} \wedge \overbrace{\{\neg hastoken(f) \wedge \neg dirty(f)\}}^{scope} \wedge$
$\overbrace{\neg \underline{Holding\ Fork(fork_f)}}^{resetting}$; *(from* **[R2]***)*

phil 3 ([*receive token$_f$ from token$_f$ avail*) $\overbrace{\underline{reqf(token_f)}}^{post-condition} \wedge \overbrace{\{hastoken(f)\}}^{scope} \wedge$
$\overbrace{\neg \underline{token_f\ avail(token_f)}}^{resetting}$; *(from* **[R3]***)*

phil 4 ([*receive fork$_f$ from fork$_f$ avail*) $\overbrace{\underline{Holding\ Fork(fork_f)}}^{post-condition} \wedge \overbrace{\{hasfork(f) \wedge \neg dirty(f)\}}^{scope} \wedge$
$\overbrace{\neg \underline{fork_f\ avail(fork_f)}}^{resetting}$; *(from* **[R4]***)*

phil 5 ([*Eating timeout(signal) : Start Thinking*) $\overbrace{\underline{Thinking(p)}}^{post-condition} \wedge \overbrace{\underline{\neg Eating(p)}}^{resetting}$; *(from* **[R5]***)*

phil 6 ([*Thinking timeout(signal) : Become Hungry*) $\overbrace{\underline{Hungry(p)}}^{post-condition} \wedge$
$\overbrace{\neg \underline{Thinking(p)}}^{resetting}$; *(from* **[R6]***)*

phil 7 ([*fork(l)*$\wedge$*fork(r) : Start Eating*) $\overbrace{\underline{Eating(p)}}^{post-condition} \wedge \overbrace{dirty(l) \wedge dirty(r)}^{scope} \wedge$
$\overbrace{\neg \underline{Hungry(p)}}^{resetting}$; *(from* **[R7]***)*

Safety Properties

phil 8 $\overbrace{(\underline{req_f(f)} \wedge \neg \underline{Holding\ Fork(token_f)} \wedge \underline{\neg Eating(p)} \wedge \underline{\neg Thinking(p)})}^{preconditions} \rightarrow (\bigcirc Occ(send\ token_f\ to\ req\ fork_f);$
(from **[R1]***)*

phil 9 $\overbrace{(\underline{Holding\ Fork(fork_f)} \wedge \underline{\neg Eating(p)} \wedge dirty(f) \wedge hastoken(f))}^{preconditions} \rightarrow (\bigcirc Occ(send\ fork_f\ to\ release\ fork_f);$
(from **[R2]***)*

phil 10 $\overbrace{(\underline{token_f\ avail(token_f)})}^{precondition} \rightarrow (\bigcirc Occ(receive\ token_f\ from\ token_f\ avail);$ *(from* **[R3]***)*

phil 11 $\overbrace{(\underline{fork_f\ avail(fork_f)})}^{precondition} \rightarrow (\bigcirc Occ(receive\ fork_f\ from\ fork_f\ avail);$ *(from* **[R4]***)*

phil 12 $\overbrace{(\underline{Eating(p)} \wedge Eating\ timeout(signal))}^{precondition} \rightarrow (\bigcirc Occ(Start\ Thinking);$ *(from* **[R5]***)*

phil 13 ($\overbrace{Thinking(p)}^{precondition} \wedge Thinking\ timeout(signal)) \rightarrow (\bigcirc Occ(Become\ Hungry)$; *(from* **[R.6]***)*

phil 14 ($\overbrace{Hungry(p)}^{precondition} \wedge fork(l) \wedge fork(r)) \rightarrow (\bigcirc Occ(Start\ Eating)$; *(from* **[R.7]***)*

Liveness Properties

phil 15 $token_f\ avail(token_f) \rightarrow \Diamond\ Occ(send\ fork_f\ to\ release\ fork_f)$

Initialization

phil 16 $Thinking(p)$

end.

In this particular example of Figure 5, the philosopher may initially possesses both his left and right fork (and they are dirty) and consequently does not have the left and right tokens. This would lead to the extra initialization formula:

$Holding\ Fork(fork_l) \wedge Holding\ Fork(fork_r) \wedge dirty(l) \wedge dirty(r)$

We assume that all predicates not mentioned in the initialization segment are initially unset.

3.4 Philosopher Local Properties

Having specified the philosopher component, we now proceed to derive local properties of its behaviour. The philosopher may exhibit many interesting properties. For example, we expect the philosopher to behaviour (ideally) in such way that there will be a cycle of activities from thinking to hungry to eating and back to thinking state.

Let us first demonstrate that if the philosopher is thinking and provided that a certain "thinking timeout" has expired, then there will be a successor slice in which he will be hungry.

T 1 $Thinking(p) \wedge Thinking\ timeout(signal) \rightarrow \bigcirc Hungry(p)$

Proof of **T 1**:

1	$Thinking(p) \wedge Thinking\ timeout(signal)$	Assumption
2	$Thinking\ timeout(signal)$	1, $\wedge_{elim}$
3	$\bigcirc Occ(Become\ Hungry)$	Phil 13, 1 MP
4	$[Become\ hungry\ ) Hungry(p) \wedge \neg Thinking(p)$	(Phil 6 ∧ 2) , ax 15 MP
5	$\bigcirc(Hungry(p) \wedge \neg Thinking(p))$	3, 4, DR 2 + temp. reas.
6	$\bigcirc Hungry(p)$	5 $\bigcirc_{\wedge}$ elim
7	$Thinking(p) \wedge Thinking\ timeout(signal) \rightarrow \bigcirc Hungry(p)$	1, 6 Discharge Assumption

Q.E.D

Now let us demonstrate that if the philosopher is eating and provided that a certain "eating timeout" has expired, then there will be a successor slice in which he will be thinking.

T 2 $\underline{Eating(p)} \wedge Eating\ timeout(signal) - \circ \underline{Eating(p)}$

Proof of **T 2**:

1	$\underline{Eating(p)} \wedge Eating\ timeout(signal)$	Assumption
2	$Eating\ timeout(signal)$	1 $\wedge_{elim}$
3	$\bigcirc Occ(Start\ Thinking)$	**Phil 12** , 1 **MP**
4	$[\ Start\ Thinking\ \rangle(\underline{Thinking(p)} \wedge \neg \underline{Eating(p)})$	**(Phil 5** $\wedge$ **2)** , ax 15 **MP**
5	$\bigcirc(\underline{Thinking(p)} \wedge \neg \underline{Eating(p)})$	3, 4 **DR2** + temp. reas.
6	$\bigcirc \underline{Thinking(p)}$	5 $\bigcirc \wedge$ elim
7	$\underline{Eating(p)} \wedge Eating\ timeout(signal) \rightarrow \bigcirc \underline{Eating(p)}$	1, 6 Discharge Assumption

Q.E.D

Several other properties are also easily derived. For example it should not be difficult to demonstrate :

T 3 $Eating(p) \wedge hasfork(f) \rightarrow dirty(f)$

4 Hygienic Dining Philosophers System Specification and Analysis

In the previous section we presented the specification of a philosopher component. We are now able to specify and reason about any system which is build upon this component. In particular let us describe a "Diner System" comprised of three philosopher components named "*plato*", "*karl*" and "*max*" which are sitting around a table, with *plato* next to *karl*, who is next to *max*, who in turn is next to *plato* (see Figure 6 below).

4.1 Informal View

A "diner system" consisting of three "philosopher" components named "*plato*", "*karl*" and "*max*" is specified by instantiating three "philosopher components" to the respective names and linking their respective ports according to the desired configuration (Figure 7)). Observe that each philosopher has a left and right neighbour philosopher, with whom he may exchange messages. Hence, the forks are shared according to the the following distribution:

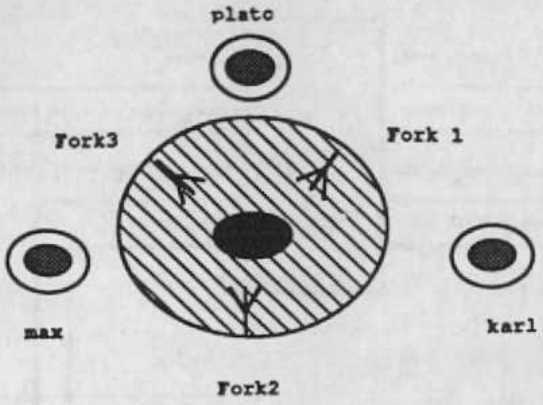


Figure 6: Diner Configuration

- philosopher *plato*: left fork = Fork1, right fork = Fork3;
- philosopher *karl*: left fork = Fork2, right fork = Fork1;
- philosopher *max*: left fork = Fork3, right f = Fork2;

Note that this intended configuration implies that the "left fork" of any philosopher is equal to the "right fork" of its neighbouring philosopher, and vice versa, i.e. neighbouring philosophers share forks. Given that, the diner system can be depicted as in Figure 7.

Note that "*plato*", "*karl*" and "*max*" are components of type philosopher. The following initial conditions are required:

- all forks are dirty
- forks are distributed among philosophers such that the precedence graph is acyclic
- if u and v are neighbours then either u holds the fork and v the priority request token or vice versa.

Let us assume that the initial distribution of forks is the following: "*plato*" possesses his left and right fork, "*karl*" does not have any fork and "*max*" possesses his right fork. All the forks are initially dirty. Moreover, "*plato*" does not possess any token, "*karl*" has both his left and right token and "*max*" possesses only his left token. Given this initial conditions and Figure 7 we can derive the following textual description.

System diner;

use Philosopher.

create

plato($fork_l, fork_r$): Philosopher;
karl($\neg fork_l, \neg fork_r$): Philosopher;
max($\neg fork_l, fork_r$): Philosopher.

link

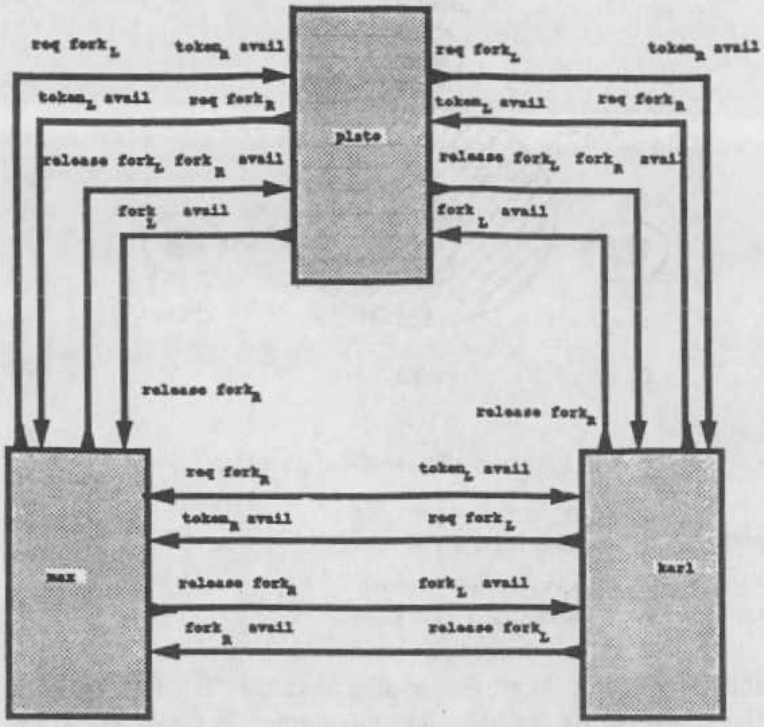


Figure 7: Diner System : Informal View

plato.req $fork_L$ to karl.$token_R$ avail;
plato.rel $fork_L$ to karl.$fork_R$ avail;
karl.req $fork_R$ to plato.$token_L$ avail;
karl.release $fork_R$ to plato.$token_L$ avail;

karl.req $fork_L$ to max.$token_R$ avail;
karl.rel $fork_L$ to max.$fork_R$ avail;
max.req $fork_R$ to karl.$token_L$ avail;
max.release $fork_R$ to karl.$token_L$ avail;

max.req $fork_L$ to plato.$token_R$ avail;
max.rel $fork_L$ to plato.$fork_R$ avail;
plato.req $fork_R$ to max.$token_L$ avail;
plato.release $fork_R$ to max.$token_L$ avail;

end.

behaviour (below)

Observe that the system has no interface ports. The **use** construct identified the type of the component used, namely, *Philosopher*. The **create** construct defines three instances of them, named "*plato*", "*karl*" and "*max*" respectively. The **link** construct defines the configuration of the various ports of the diner system.

The behaviour of the diner system is defined by the causal and temporal-causal view given below.

4.2 Enhanced Net Composition (Causal View)

Again, the causal view is obtained from the diner structural configuration specification (see Figure 8). Recall that each philosopher component could be thought of as a transition whose typed output/input places corresponded to the typed exit/entry ports of the philosopher informal view. The task of composing philosopher components into diner system specifications is straightforward: the output/input places of a philosopher are joined together (i.e. are identified) with the input/output places of a neighbouring philosopher, according to the "configuration" (links) presented in the informal view of Figure 7.

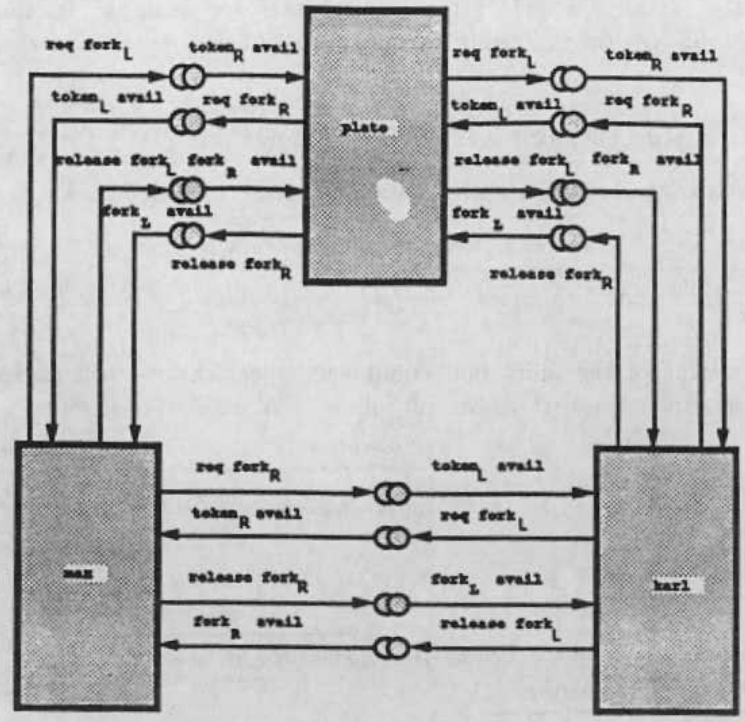


Figure 8: Diners System: Causal View

Hence, the following interface places are identified:

- plato.$req\ fork_L$ ($token$) is equal to karl.$token_R\ avail$ ($token$);
- plato.$rel\ fork_L$ ($fork$) is equal to karl.$fork_R\ avail$ ($fork$);
- karl.$req\ fork_R$ ($token$) is equal to plato.$token_L\ avail$ ($token$);
- karl.$release\ fork_R$ ($fork$) is equal to plato.$token_L\ avail$ ($fork$);
- karl.$req\ fork_L$ ($token$) is equal to max.$token_R\ avail$ ($token$);
- karl.$rel\ fork_L$ ($fork$) is equal to max.$fork_R\ avail$ ($fork$);

- max.$\underline{req\ fork_R}$ ⟨ token ⟩ is equal to karl.$\underline{token_L\ avail}$ ⟨ token ⟩;
- max.$\underline{release\ fork_R}$ ⟨ fork ⟩ is equal to karl.$\underline{token_L\ avail}$ ⟨ fork ⟩;
- max.$\underline{req\ fork_L}$ ⟨ token ⟩ is equal to plato.$\underline{token_R\ avail}$ ⟨ token ⟩;
- max.$\underline{rel\ fork_L}$ ⟨ fork ⟩ is equal to plato.$\underline{fork_R\ avail}$ ⟨ fork ⟩;
- plato.$\underline{req\ fork_R}$ ⟨ token ⟩ is equal to max.$\underline{token_L\ avail}$ ⟨ token ⟩;
- plato.$\underline{release\ fork_R}$ ⟨ fork ⟩ is equal to max.$\underline{token_L\ avail}$ ⟨ fork ⟩;

4.3 Temporal-Causal View

The intended configuration defines three instances (*plato*, *karl* and *max*) of the philosopher component. We adhere to the notational device presented above, i.e. prefix any syntactic categories by the instance name of the component. For example, in the diner system example, there will be the following interface condition predicates (where f can take value r or l):

- $\underline{plato.req\ fork_f}$, $\underline{karl.req\ fork_f}$, $\underline{max.req\ fork_f}$: ⟨token⟩;
- $\underline{plato.token\ avail_f}$, $\underline{karl.token\ avail_f}$, $\underline{max.token\ avail_f}$: ⟨token⟩;
- $\underline{plato.release\ fork_f}$, $\underline{karl.release\ fork_f}$, $\underline{max.release\ fork_f}$: ⟨fork⟩;
- $\underline{plato.fork_f\ avail}$, $\underline{karl.fork_f\ avail}$, $\underline{max.fork_f\ avail}$: ⟨fork⟩;

Moreover, axioms of the individual component specifications will also use this "dot" notation. For example, the first axiom of "plato" philosopher component specification:

$$\text{Phil 1}\ ([\ send\ token_f\ to\ req\ fork_f\ \rangle \overbrace{\underline{req\ fork_f(token_f)}}^{post-condition} \wedge (\overbrace{\neg hastoken_f(f)}^{scope})) \wedge \overbrace{\neg \underline{reqf(f)}}^{resetting};$$

now corresponds to:

$$\text{plato.Phil 1}\ ([\ plato.send\ token_f\ to\ req\ fork_f\ \rangle \overbrace{\underline{plato.req\ fork_f(token_f)}}^{post-condition} \wedge (\overbrace{\neg plato.hastoken_f(f)}^{scope})) \wedge \overbrace{\neg \underline{plato.reqf(f)}}^{resetting};$$

For each "link" associated with the previous views, there will be a corresponding temporal-causal configuration axiom. For example, from Figure 8 we can identify twelve configuration axioms:

Diner 1 plato.$\underline{req\ fork_L}$ ⟨ $token_l$ ⟩ ↔ karl.$\underline{token_R\ avail}$ ⟨ $token_r$ ⟩;

Diner 2 plato.$\underline{rel\ fork_L}$ ⟨ $fork_l$ ⟩ ↔ karl.$\underline{fork_R\ avail}$ ⟨ $fork_r$ ⟩;

Diner 3 karl.$\underline{req\ fork_R}$ ⟨ $token_r$ ⟩ ↔ plato.$\underline{token_L\ avail}$ ⟨ $token_l$ ⟩;

Diner 4 karl.$\underline{release\ fork_R}$ ⟨ $fork_r$ ⟩ ↔ plato.$\underline{token_L\ avail}$ ⟨ $fork_l$ ⟩;

Diner 5 karl.$\underline{req\ fork_L}$ ⟨ $token_l$ ⟩ ↔ max.$\underline{token_R\ avail}$ ⟨ $token_r$ ⟩;

Diner 6 karl.$\underline{rel\ fork_L}$ ⟨ $fork_l$ ⟩ ↔ max.$\underline{fork_R\ avail}$ ⟨ $fork_r$ ⟩;

Diner 7 max.$\underline{req\ fork_R(\ token_r\ )}$ ← karl.$\underline{token_L\ avail(\ token_l\ )}$;

Diner 8 max.$\underline{release\ fork_R(\ fork_r\ )}$ ← karl.$\underline{token_L\ avail(\ fork_l\ )}$;

Diner 9 max.$\underline{req\ fork_L(\ token_l\ )}$ ← plato.$\underline{token_R\ avail(\ token_r\ )}$;

Diner 10 max.$\underline{rel\ fork_L(\ fork_l\ )}$ ← plato.$\underline{fork_R\ avail(\ fork_r\ )}$;

Diner 11 plato.$\underline{req\ fork_R(\ token_r\ )}$ ← max.$\underline{token_L\ avail(\ token_l\ )}$;

Diner 12 plato.$\underline{release\ fork_R(\ fork_r\ )}$ ← max.$\underline{token_L\ avail(\ fork_l\ )}$;

The initial conditions of the three philosophers will account for three extra temporal-causal axioms.

Diner 13 $\underline{plato.Holding\ \ Fork(fork_l)}\wedge\underline{plato.Holding\ \ Fork(fork_r)}\wedge plato.dirty(l)$
$\wedge plato.dirty(r)$;

Diner 14 $\underline{karl.reqf(token_l)}\wedge\underline{plato.reqf(token_r)}\wedge karl.hastoken(l)\wedge karl.hastoken(r)$;

Diner 15 $\underline{max.Holding\ \ Fork(fork_r)}\wedge\underline{max.reqf(token_l)}\wedge max.dirty(r)\wedge max.hastoken(l)$;

4.4 Hygienic Diner System Global Properties

Having specified the system, we may now want to derive the global properties of it. Of course we do not have to redo the proofs of the local components, because the temporal-causal framework is compositional (see [4]).

The initial diner system configuration guarantees that:

- all forks are dirty $(plato.dirty(l)\wedge plato.dirty(r)\wedge max.dirty(r))$;
- every fork and request fork are held by different philosophers:
  - $\underline{plato.Holding\ \ Fork(fork_l)}\wedge\underline{karl.reqf(token_r)}$
  - $\underline{plato.Holding\ \ Fork(fork_r)}\wedge\underline{max.reqf(token_l)}$
  - $\underline{max.Holding\ \ Fork(fork_r)}\wedge\underline{karl.reqf(token_l)}$
- all forks are located at philosophers in such a way that H (precedence graph) is acyclic.

Furthermore, the directions of edges in H may be affected only when a fork changes its status (dirty or clean) or its location. But every change to H preserves acyclicity. Recall from **Phil 2**, that every transmission of fork is accompanied by a change in its status from dirty to clean, but this does not change the direction of any edge. From **Phil 7** we conclude that a fork is dirtied when the philosopher u holding it, eats. But this is a guarded event which requires that the philosopher possesses all forks associated with edges incident upon it. Therefore u cannot create a cycle in H because all edges upon u are directed toward it. Consequently **H is always acyclic.** Furthermore, immediately upon completion of an eating session a philosopher yields precedence to his neighbour.

We want to prove that every hungry philosopher will eat.

T 4 $\underline{Hungry(p)}\rightarrow\bigcirc\underline{Eating(p)}$

As expected, Chandy and Misra solution and proofs can be translated easily to our framework. This is done as follows:

The formal proof of this property is based upon the fact that a hungry philosopher, which is not in possession of both forks (consequently has the tokens) will request the forks to its neighbour (**Phil 8**), which will eventually grant it (axiom **Phil 15**). And since the fork is clean upon receipt (**Phil 4**), the philosopher will hold it until he eats (from **Phil 7** and **Phil 8**). A philosopher requesting a fork that is clean must make the request to a philosopher at a smaller depth and, by induction on depth, this philosopher will eat and then dirty the fork, in which case the first argument applies.

The depth of a philosopher in H is the maximum number of edges along a path to that philosopher from one without predecessors. A hungry philosopher at depth 0 in H will commence eating in finite time (because he has precedence over all his neighbours). By induction on depth, a hungry philosopher at depth $k, k \geq 0$, will eat in finite time because he has precedence over all philosophers at greater depth, and all philosophers at smaller depth will yield precedence to it in finite time.

Let u, v be neighbours and u be hungry. We can show that u holds or will hold the fork f corresponding to the edge (u, v) and will thereafter continue to hold it until u eats. If u holds the fork currently and holds it continuously until he eats, the result is trivial. Therefore assume that v holds the fork f sometime before u eats next. At this slice we have:

$$\neg u.\underline{Eating(p)} \wedge \neg u.\underline{Thinking(p)} \wedge \neg u.hasfork(f) \wedge v.hasfork(f).$$

case 1: f is dirty ($v.dirty(f)$), If $u.\overline{reqf(token_f)}$ holds then u will request f (**Phil 8**) and subsequently $v.\overline{reqf(token_f)}$ will hold (from **Phil 10** and **Phil 3**); otherwise $v.\overline{reqf(token_f)}$ already holds. If $v.\overline{Eating(p)}$ holds then at some later point (since eating is finite), $\neg v.\overline{Eating(p)}$ (from **Phil 5**) and all conditions for rule **Phil 9** still hold. Therefore **Phil 2** will be applied to v, and u will eventually hold a clean fork u. u will not release a clean fork until u eats.

Case 2: f is clean $\neg v.dirty(f)$), Every fork held a a nonhungry philosopher is dirty because:

- all forks are dirty initially (**Initialization**),
- only hungry philosophers receive clean forks (**Phil 4**), and
- all forks held by eating philosophers are dirty (**Phil 7**).

Since f is clean, the philosopher v holding it must be hungry. Furthermore, because f is clean, (v, u) is an edge in H and hence $depth(v) < depth(u)$. According to the induction hypothesis, v eats and hence dirties f. Case 1 then applies.

5 Conclusions

We argued that a single representation scheme is often not enough to capture the various features of system behaviour. Instead, multiple viewpoints should be used to partition the domain of information. Its success rests upon the selection of appropriate representation schemes, the careful definition of the relations between them and the process by which such specifications are built within those representation schemes.

We presented a technique that supports the specification of systems from three points of view: informal. causal and temporal-causal. The *Informal View* describes in natural language and graphically the requirements of the component. It is based on the *Conic Environment* which provides a language-based approach to the building of distributed system. The *Causal View* relies on enhanced Petri Nets to describe internal structure, distributed control and safety requirements. The third view makes use of temporal-causal logic to describe temporal features of the components such as liveness requirements. Previous causal properties are preserved. Temporal-causal logic is the result of the integration of two well-known formalisms, namely Petri Nets and Temporal Logic.

In this paper showed that the framework can be used to describe the behaviour of distributed systems and to reason about it. In particular we are able to prove local properties of the components behaviour as well as global properties of the system composed of separate, interacting *components*.

6 Acknowledgements

I gratefully acknowledge CNPq, under grant 301052/91-3(NV) for its financial support. I also acknowledge the usefull discussions with Jeff Kramer at Imperial College.

References

[1] J. Castro. A Pluralistic Approach to Distributed System Specification extended abstract. Second IEEE Symposium on Parallel & Distributed Processing, December 1990. Dallas, United States.

[2] J. Castro and J. Kramer. Constructing Distributed System Specification: A Temporal-Causal Approach. In *Proceedings of X Congress of The Brazilian Computing Society*, pages 106–120, July 1990. Vitória, Brazil.

[3] J. Castro and J. Kramer. Temporal-Causal System Specifications. In *Proceedings of IEEE International Conference on Computer Systems and Software Engineering (CompEuro90)*, pages 210–217, May 1990. Tel-Aviv, Israel.

[4] Jaelson F. B. Castro. *Distributed System Specification Using A Temporal-Causal Framework*. PhD thesis, University of London, October 1990. Imperial College of Science, Technology and Medicine, Department of Computing.

[5] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM TOPLAS*, 6(4):632–646, October 1984.

[6] J. Kramer. Configuration Programming - A Framework for the Development of Distributable Systems. In *Proceedings of IEEE International Conference on Computer systems and software Engineering (CompEuro90)*, pages 374–384. IEEE Computer Society Press, May 1990.

[7] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. In *IEEE Transaction on Software Engineering*, page 35pp, 1990.

[8] Wolfgang Reisig. Temporal logic and causality in concurrent systems. In F. H. Vogt, editor, *Proceedings of Concurrency 88*, pages 121–139. Springer-Verlag, 1988. LNCS 335.

[9] Wolfgang Reisig. Towards a temporal logic for causality and choice in distributed systems. In J.W Bakker, W. P. de Roever, and G. Rozemberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 603–627. Springer-Verlag, 1989. LNCS 354.

[10] Ichiro Suzuki and HarnGdar Lu. Temporal petri nets and their application to modeling and analysis of a handshake daisy chain arbiter. *IEEE Transactions on Computers*, 38(5), May 1989.