A Technique for Specifying and Refining TCSP Processes by Using Guards and Liveness Conditions

R. Peña
Departamento de Informática y Automática
Universidad Complutense de Madrid
E-28040 Madrid. Spain
e-mail: ricardo@dia.ucm.es

Luis M. Alonso
Departamento de Lenguajes y Sistemas Informáticos
Universidad del Pais Vasco
E-20080 San Sebastian. Spain
e-mail: alonso@gorria.if.ehu.es

Abstract

A technique for the specification of TCSP processes based upon the concepts of *guards* and *liveness rules* is presented. It is shown how safety and liveness properties can be proved for processes specified in this way. A technique related to bisimulations is proposed to prove refinements correct. The technique is extended to handle the concealment of events in the implementing process. The refinement relation preserves the safety and liveness properties already proved for the specification. Parallel composition of specifications is also defined preserving the failures semantics. To illustrate the technique, an example is used throughout the paper.

1 Introduction

The *failures* model [5,6,9] has proved to be a successful theoretical and practical tool for the specification and verification of parallel systems. Other related algebraic theories and calculus of processes are those by Milner [12,13], Hennessy [8], and Bergstra and Klop [4]. The notion of *trace* of the process, i.e. the sequence of actions already performed by the process, plays in TCSP a central role. The so called *refusal sets* provide a means to define the inmediate liveness of the process after a given trace.

Starting with these concepts, the authors proposed in [15] a technique for TCSP process specification. Valid traces were defined by partial abstract types, as described in [7]. These ideas evolved in [2,3] where the notion of state for defining TCSP processes

was made explicit in the form of *state variables*. States corresponding to legal traces were characterized by means of *invariants*, much in the sense of data representation invariants defined in [10].

The present paper represents the culmination of these ideas. Its main contribution is a change in the technique for building specifications and proving refinements correct. State variables are retained, but now process behaviour is specified by two sets of conditions: *safety* requirements and *liveness* requirements. The first one takes the form of a family of *guards* and defines the allowed traces for the process. The second one consists of a collection of set expressions, defining the mandatory events in each state.

Specifications with this technique are shorter and more abstract than those presented in the previous papers. Proving refinements correct and proving that a specification satisfies a liveness predicate, are also easier.

The organization of the paper is as follows: Section 2 defines the concept of *process specification with safety and liveness conditions*, first by using an example and then formally. Its semantics is given in terms of the failures model. Section 3 explains how safety and liveness properties can be proved for processes specified in this way. In Section 4 we characterize the notion of refinement of a process by another. Concealment of events in the implementing process is also considered. In Section 5, we define the parallel composition of process specifications. Finally, Section 6 provides a short conclusion. In this paper, we provide neither the proofs of the propositions nor meaningful examples. The interested readers are addressed to [1].

2 Process Specification Using Safety and Liveness Conditions

The technique used in [9] for specifying processes consisted of a predicate $S(tr, X)$ with free variables tr (for traces) and X (for *refusal sets*). A process is said to satisfy a specification, denoted P **sat** $S(tr, X)$, if every legal trace tr of P and every refusal set X of P after engaging in the trace tr, satisfy the predicate $S(tr, X)$, i.e.:

$$P \textbf{ sat } S(tr, X) \triangleq \forall (tr, X) \in P.S(tr, X)$$

Here, we are assuming that a non divergent *TCSP* process is a subset of $\mathcal{L}^* \times \mathcal{P}(\mathcal{L})$, $\mathcal{L}$ being the alphabet of the process. A trace tr represents a possible event history for P and a refusal X represents a menu of events such that P *may* deadlock if the external environment of P offers that menu.

Also in [9], a set of proof rules for verifying the **sat** relation was given. These are based on the syntactic structure of P and take the form of deduction rules such as the following one:

$$P \textbf{ sat } S(tr, X) \wedge Q \textbf{ sat } T(tr, X) \Rightarrow P \sqcap Q \textbf{ sat } S(tr, X) \vee T(tr, X)$$

where $\sqcap$ is the internal choice *TCSP* operator. In case P is recursive, some form of induction is needed to verify the desired property. In general, the proofs of realistic specifications using these rules tend to be very hard and they have an *ad hoc* look as they heavily depend on the specific syntax of the involved processes.

In this section we propose both a technique for defining *TCSP* processes using a syntactical *normal form*, and a method for proving the sat relation in a more systematic way. In fact, the technique can be seen as a way of structuring the $S(tr, X)$ predicate according to some simple rules, giving as result a process satisfying the predicate.

First, it is worthy to note that $S(tr, X)$ can be split into two predicates, one on traces establishing which are the allowed traces for the process, and another one establishing the *future* of the process after a legal trace. In the *TCSP* jargon, it is traditional to say that the first one specifies the *safety* properties of the system while the second one expresses its *liveness* properties, i.e.,

$$S(tr, X) = Safe(tr) \wedge Live(tr, X)$$

Example 1 A mutual exclusion arbiter for n users
Let us assume a system in which n users synchronize with an arbiter by means of the events req_i (user i asks for permission to use the resource), ack_i (the arbiter gives the permission to user i) and rel_i (user i has finished using the resource). The safety predicate for a robust arbiter can be defined as follows:

$$\begin{aligned} &Safe(tr) \triangleq \\ &\quad \forall i \in U.\#(tr{\uparrow}rel_i) \leq \#(tr{\uparrow}ack_i) \leq \#(tr{\uparrow}req_i) \leq \#(tr{\uparrow}rel_i) + 1 \\ &\quad \wedge \\ &\quad (Ni : i \in U.\#(tr{\uparrow}rel_i) < \#(tr{\uparrow}ack_i)) \leq 1 \end{aligned} \tag{1}$$

where $tr{\uparrow}e$ means the projection of the trace tr over the alphabet $\{e\}$, N is the counting quantifier, $\#$ denotes the number of events of a trace and $U = \{1 \ldots n\}$.

Another way of specifying $Safe(tr)$ is to define it as the following inductively generated set of traces:

- $Safe(<>)$
- $Safe(tr) \wedge thinking_i(tr) \Rightarrow Safe(tr^\frown req_i)$
- $Safe(tr) \wedge eating_i(tr) \Rightarrow Safe(tr^\frown rel_i)$
- $Safe(tr) \wedge hungry_i(tr) \wedge \neg eaters(tr) \Rightarrow Safe(tr^\frown ack_i)$

where

$$\begin{aligned} thinking_i(tr) &\triangleq \#(tr{\uparrow}req_i) = \#(tr{\uparrow}rel_i) \\ hungry_i(tr) &\triangleq \#(tr{\uparrow}ack_i) < \#(tr{\uparrow}req_i) \\ eating_i(tr) &\triangleq \#(tr{\uparrow}rel_i) < \#(tr{\uparrow}ack_i) \\ eaters(tr) &\triangleq \exists i \in U.eating_i(tr) = true \end{aligned}$$

That is, for every legal trace tr and every event e, we state the boolean condition $G_e(tr)$ under which it is *safe* to allow the event e to be added to tr. In what follows, we will use the term *guard* when referring to those conditions G_e.

This way of defining $Safe(tr)$ gives us directly an operational definition of a safe process: let us imagine a process having, as internal state, a variable of type *trace*

process $MutEx_s$
  alphabet req_i, ack_i, rel_i
  state variables
   $s : \mathcal{L}^* := <>$
  transition rules
   on $\text{req}_i \Rightarrow s := s^\frown req_i$
   on $\text{ack}_i \Rightarrow s := s^\frown ack_i$
   on $\text{rel}_i \Rightarrow s := s^\frown rel_i$
  requirements
   $\text{req}_i \Rightarrow thinking_i(s)$
   $\text{ack}_i \Rightarrow hungry_i(s) \wedge \neg eaters(s)$
   $\text{rel}_i \Rightarrow eating_i(s)$
end process $MutEx_s$

Figure 1: Chaotic mutual exclusion arbiter specified by using traces

where it *stores* the trace performed by it up to a particular moment. It can use the guards G_e (which are boolean functions on traces) to know the set of safe events in which it can engage at that moment. It then makes a nondeterministic choice and decides either to engage in one of these events or to stop. If it decides to engage in one event, then updates its internal state, recording that this event is now part of the trace (in fact it is the last event). The description of such a process for example 1 is shown in figure 1. The boolean condition after the expression $e\Rightarrow$ in the "requirements" section, is the guard G_e. It is understood that $i \in U$ and that a free i in a line means the replication of that line for all $i \in U$.

This kind of process with internal memory starts its execution with the empty trace as initial state and, in every state, it behaves like what we can call a *safe chaotic* process, in short *safe chaos*. It is the most nondeterministic process of those that satisfy $Safe(tr)$. Every safe process is *included* in it.

If we have the predicate $Safe(tr)$ explicitly defined and we want to prove that a process P built with guards satisfies it, it is enough to think of $Safe(tr)$ as an invariant of P. We must prove that $Safe(tr)$ holds for the empty trace and that it is preserved by every safe transition. This technique is explained in detail in section 3.

We turn now to the liveness conditions. The **stop** process and, in general, any process that stops after executing a legal trace, are included among the safe ones. Usually, we would like to require a process to accept certain (safe) events in some states. In *TCSP*, *refusal sets* are used to impose obligations on a process. In [2,3] the authors used a (perhaps more explicit) *menu relation* to specify the set of nondeterministic menus offered by the process in any state. The translation from menus to refusals is immediate. If a particular event is included in *all* the menus associated to a particular state, then the process is forced to engage in this event if the environment insists. In other words, the event is deterministic. Otherwise, it is nondeterministic. Of course, there must be a consistency between the safety and the liveness conditions imposed on a process.

Here, we have chosen to define the obligations of a process by means of *liveness*

rules. A liveness rule is a (total) function from states to sets of events. Its semantics, formally given below, is that in every state the process must be able to engage in at least one of the events of (the evaluation of) the liveness rule in that state. If there exist several liveness rules, the process must satisfy the obligations imposed by all of them.

The mutual exclusion arbiter with liveness rules can be built by adding to figure 1 the following liveness rules:

$$\begin{aligned} \forall i \in U.Req_i(s) &\triangleq \{req_i \mid thinking_i(s) = true\} \\ \forall i \in U.Rel_i(s) &\triangleq \{rel_i \mid eating_i(s) = true\} \\ Ack(s) &\triangleq \{ack_i \mid hungry_i(s) \wedge \neg eaters(s)\} \end{aligned}$$

In this specification we require that the events req_i and rel_i be deterministic whenever user i is respectively *thinking* or *eating.* Also, one of the events ack_i for all *hungry* users i, is compulsory in states such that no user is *eating.* When an event e is both safe and compulsory in a state, we will use the abbreviation $e \Leftrightarrow G_e$. This is equivalent to defining the guard G_e and the following liveness rule

$$\text{if } G_e(tr) \text{ then } \{e\} \text{ else } \emptyset$$

A specification with guards and liveness rules defines, as we will see, a unique process in the failures model. Proving that it satisfies an explicit predicate $S(tr, X) = Safe(tr) \wedge Live(tr, X)$ is a straightforward task (see section 3). The advantage of the proposed method is that we have not only built an abstract specification, but also an actual system to start the design process with. The rest of the task consists of *refining* the system and proving that the refinements are correct.

Before proceeding to the formal definitions, let us now introduce the concept of *state variables* as defined in [3]. In the example of figure 1, it is obvious that the information kept by the process in its internal memory is excessive. In most of the examples, to record the relevant data about the past of the process, it is enough to have a finite set of "small" variables. Based on this information, the process can take exactly the same decisions as if it had the complete trace stored. In the example, the only variables that are needed are the individual states st_i of the users, where:

$$st_i : UserState \quad \text{and} \quad UserState \triangleq \{thinking, hungry, eating\}$$

For all $s \in \mathcal{L}^*$, we want to preserve the following invariant relation:

$$\begin{aligned} Safe(s) \;\Rightarrow\; & \forall i \in U. \\ & thinking_i(s) \Leftrightarrow st_i = thinking \\ \wedge\; & hungry_i(s) \Leftrightarrow st_i = hungry \\ \wedge\; & eating_i(s) \Leftrightarrow st_i = eating \end{aligned} \tag{2}$$

The translation of the arbiter specified with traces to an arbiter specified with state variables st_i, is shown in figure 2. There, the predicate *eaters* can be defined in terms of the new state variables as:

$$(Ni.st_i = eating) > 0 \tag{3}$$

process $MutEx$
  alphabet req_i, ack_i, rel_i
  state variables
   st_i: UserState := *thinking*
  transition rules
   on $\text{req}_i \Rightarrow st_i$:= *hungry*
   on $\text{ack}_i \Rightarrow st_i$:= *eating*
   on $\text{rel}_i \Rightarrow st_i$:= *thinking*
  requirements
   $\text{req}_i \Leftrightarrow st_i$ = *thinking*
   $\text{ack}_i \Rightarrow (st_i = hungry) \wedge \neg eaters$
   $\text{rel}_i \Leftrightarrow st_i = eating$
   $Ack \triangleq \{ack_i \mid st_i = hungry \wedge \neg eaters\}$
end process $MutEx$

Figure 2: Mutual exclusion arbiter with state variables st_i

We regard the state variables of a process as *observer* functions on traces. Each state variable conveys some relevant information about the past history of the process. The transition rules can be looked at as the definitions of these observer functions. They explain how the observation changes as we concatenate a new event to the trace. Let us note that these observer functions are partial functions over $\mathcal{L}^*$ and total ones over $traces(P)$. The same happens to the guards and liveness rules since they are expressed in terms of the primitive observers st_i.

Now we proceed to the formal definitions. For the rest of the paper, we assume the existence of some predefined data domains $\mathcal{D}_{t_1} \ldots \mathcal{D}_{t_n}$ with type names $t_1 \ldots t_n$.

We will use a tuple $x_1 \ldots x_n$ of *state variables* to represent the set of states. Any state σ may be seen as an assignment $\{x_1 \leftarrow \kappa_1, \ldots, x_n \leftarrow \kappa_n\}$ of values from the appropriate data domains to state variables. In particular, the initial state of the process is given by an *initial assignment* σ_0. From now on, given a set of state variables $\mathcal{V} = \{x_1, \ldots, x_n\}$, we will denote by $Ass(\mathcal{V})$ the set of all possible assignments $\sigma : \mathcal{V} \longrightarrow \mathcal{D}$ of values to variables in $\mathcal{V}$.

The state transitions are described giving expressions $E_{e,x}$ for every pair of event e and state variable x. If $\sigma \xrightarrow{e} \sigma'$ is a transition, the value of x in σ' is defined as the value of $E_{e,x}$ in σ.

Definition 2 A *process specification with safety and liveness conditions*, in short a *process specification*, is given by a tuple $SP = (\mathcal{L}, \mathcal{V}, \sigma_0, \mathbf{TR}, \mathbf{G}, \mathbf{L})$ where:

- $\mathcal{L}$ is a nonempty *alphabet* of events
- $\mathcal{V} = \{x_1 \ldots x_n\}$ is a finite set of typed variables, called *state variables*
- σ_0, the *initial assignment*, associates to every state variable x an appropiate value, denoted $\sigma_0(x)$

- **TR** is a $\mathcal{L}$-indexed family $(TR_e)_{e\in\mathcal{L}}$ of *transition rules*. A transition rule TR_e associates to every state variable x a properly formed expression $E_{e,x}$, with free variables in $\mathcal{V}$.
- **G** is a $\mathcal{L}$-indexed family $(G_e)_{e\in\mathcal{L}}$ of boolean expressions with free variables in $\mathcal{V}$, called *guards*.
- **L** is a set $\{L_i(\mathcal{V})\}$ of expressions, with free variables in $\mathcal{V}$. The type of each of them is "set of events of $\mathcal{L}$". Each L_i is called a *liveness rule*.

Very often, expressions defining transition or liveness rules will use auxiliary functions defined in some suitable formalism. For simplicity, in the rest of the paper we shall assume that such functions are totally defined over $Ass(\mathcal{V})$. As we will immediately see, not every state in $Ass(\mathcal{V})$ is a state *reachable* by the process.

Let $\sigma \in Ass(\mathcal{V})$ and let $E(\mathcal{V})$ be an expression with free variables in $\mathcal{V}$. We will denote by $\overline{\sigma}(E(\mathcal{V}))$, in short $\overline{\sigma}(E)$, the evaluation of the expression E after assigning values to variables by σ. Let $TR_e = \{E_{e,x_1} \ldots E_{e,x_n}\}$, be a transition rule, we will denote by $TR_e \circ \sigma$ the following assignment:

$$TR_e \circ \sigma = \{x_i \leftarrow \overline{\sigma}(E_{e,x_i}), x_i \in \mathcal{V}\}$$

Last, given a predicate $\mathcal{P}$ with free variables in $\mathcal{V}$, we will denote by $TR_e(\mathcal{P})$ the predicate obtained by substituting E_{e,x_i} for every occurence of variable x_i in $\mathcal{P}$. In what follows, we shall assume the implicit existence of a process specification:

$$SP = (\mathcal{L}, \mathcal{V}, \sigma_0, \mathbf{TR}, \mathbf{G}, \mathbf{L})$$

Definition 3 The set of *reachable* states of SP, denoted Σ_{SP}, is the following inductively generated set of assignments:

1. $\sigma_0 \in \Sigma_{SP}$
2. $\forall \sigma \in \Sigma_{SP}, \forall e \in \mathcal{L}.(\overline{\sigma}(G_e) = true \Rightarrow TR_e \circ \sigma \in \Sigma_{SP})$

Definition 4 The *transition relation* of SP, denoted $\longrightarrow_{SP}$, abbreviated $\longrightarrow$, is the following inductively generated set of triples $\sigma \xrightarrow{t} \sigma' \in \Sigma_{SP} \times \mathcal{L}^* \times \Sigma_{SP}$:

1. $\forall \sigma \in \Sigma_{SP}.\sigma \xrightarrow{<>} \sigma$
2. $\forall \sigma_1, \sigma_2 \in \Sigma_{SP}, \forall t \in \mathcal{L}^*, \forall e \in \mathcal{L}.(\sigma_1 \xrightarrow{t} \sigma_2 \wedge \overline{\sigma}_2(G_e) = true \Rightarrow \sigma_1 \xrightarrow{t^\frown e} TR_e \circ \sigma_2)$

Definition 5 The set of *traces* of SP, denoted $traces(SP)$, is defined from $\longrightarrow_{SP}$ as the following set:

$$traces(SP) = \{t \in \mathcal{L}^* \mid \exists \sigma_t \in \Sigma_{SP}.\sigma_0 \xrightarrow{t} \sigma_t\}$$

We will denote by σ_t the state reached by SP after executing the trace t. Let us note that the mapping defined by $\sigma(t) = \sigma_t$ is not, in general, an injection and always is a surjection over Σ_{SP}.

Definition 6 The set of *possible* events of SP after the trace t, is defined as:

$$next(t) = \{e \in \mathcal{L} \mid \overline{\sigma}_t(G_e) = true\}$$

Definition 7 A pair (t, m), where $t \in traces(SP)$ and $m \subseteq \mathcal{L}$ *satisfies* SP, denoted (t, m) **sat** SP, if

- $m \subseteq next(t)$, and
- $\forall L \in \mathbf{L}.(\overline{\sigma}_t(L) \neq \emptyset \wedge \overline{\sigma}_t(L) \subseteq next(t) \Rightarrow m \cap \overline{\sigma}_t(L) \neq \emptyset)$

This definition expresses the semantics we want for our specifications. Let us emphasize the following aspects:

- in a state σ_t, if a liveness rule L gives rise to a non empty set of events $\overline{\sigma}_t(L)$, and all these events are safe in that state, then all the menus m of the process in state σ_t must include at least one event of $\overline{\sigma}_t(L)$.
- if all liveness rules $L \in \mathbf{L}$ give rise to empty sets in a state $\overline{\sigma}_t$, then any subset of $next(t)$, even the empty set, satisties SP. We say that SP behaves as a *safe chaos* in that state.
- if a liveness rule L imposes a partially unsafe set of events in a state, then it has no effect in that state. (This is an arbitrary decission but it has proved to be useful in the examples the authors have tried.)
- if $next(t) = \emptyset$ in a state, then the only menu satisfying SP in that state is $\emptyset$. This will be a deadlock state.
- if $next(t) - \bigcup_{L \in \mathbf{L}} \overline{\sigma}_t(L) \neq \emptyset$, all the events in this difference can be chosen by the process in a non deterministic way. They are neither mandatory, nor forbidden in that state.

Expressing this desired semantics into the failures model is immediate:

Definition 8 The *failures semantics* of SP, denoted $[\![SP]\!]$, is the following inductively generated subset of $\mathcal{L}^* \times \mathcal{P}(\mathcal{L})$:

1. $\forall t \in traces(SP), \forall m \subseteq \mathcal{L}.(t, m)$ **sat** $SP \Rightarrow (t, \overline{m}) \in [\![SP]\!]$
2. $\forall t \in traces(SP), \forall X_1, X_2 \subseteq \mathcal{L}.(t, X_1) \in [\![SP]\!] \wedge X_2 \subseteq X_1 \Rightarrow (t, X_2) \in [\![SP]\!]$

Proposition 9 Given a process specification $SP = (\mathcal{L}, \mathcal{V}, \sigma_0, \mathbf{TR}, \mathbf{G}, \mathbf{L})$, $[\![SP]\!]$ is a process, in particular a non divergent one, in the failures model.

3 Proving Properties

As it has been said, in [9] the specification of a process takes the form of a predicate with free variables tr and X denoting any trace and refusal set of the process. The expression P **sat** $S(tr, X)$, is formalized by:

$$\forall tr, X.tr \in traces(P) \wedge X \in refusals(P/tr) \Rightarrow S(tr, X)$$

In our context, safety properties can be expressed as a predicate over the process state variables, i.e. as a predicate *Safe* with free variables in $\mathcal{V}$. Given the corresponding predicate on traces, its statement using state variables is usually immediate.

To prove a safety property written in terms of state variables, it must be shown that it holds for all values of state variables corresponding to *legal* traces of the process. The assignments corresponding to reachable states must then be characterized.

Definition 10 We shall denote by $Reach_{SP}$ the strongest predicate, with free variables in $\mathcal{V}$, satisfying the following conditions:

- $\overline{\sigma}_0(Reach_{SP}) = true$
- $G_e \wedge Reach_{SP} \Rightarrow TR_e(Reach_{SP})$, holds for all $e \in \mathcal{L}$.

Then the proof of any safety property *Safe*, would reduce to prove the following implication:

$$Reach_{SP} \Rightarrow Safe$$

Usually, we do not need to know exactly $Reach_{SP}$. In most situations, it suffices proving that $Inv_{SP} \Rightarrow Safe$ holds, for some predicate Inv_{SP} weaker than $Reach_{SP}$ which is invariant in the following sense:

Definition 11 A predicate Inv, with free variables in $\mathcal{V}$, is an *SP-invariant* if:

- $\overline{\sigma}_0(Inv) = true$
- $G_e \wedge Inv \Rightarrow TR_e(Inv)$, holds for all $e \in \mathcal{L}$.

The reader can easily prove that the following predicate is an invariant of the mutual exclusion arbiter of figure 2:

$$(Ni \in U.st_i = eating) \leq 1 \tag{4}$$

We will discuss now the analysis of liveness properties. The relevant definitions are:

Definition 12 A specification SP is *well formed* if there exists an SP-invariant Inv_{SP} such that

$$Inv_{SP} \Rightarrow \forall L \in \mathbf{L}.L \subseteq next_{SP}$$

where $next_{SP}(\mathcal{V})$ is a set-of-events expression defined in terms of state variables as $\{e \mid e \in \mathcal{L} \wedge G_e(\mathcal{V})\}$. It represents the set of safe events the process can engage in after each state.

Well formedness removes the need to check in all the proofs whether the liveness requirements are in contradiction or not with the safety ones.

Definition 13 Given a well formed specification SP, an SP-invariant Inv_{SP}, and a liveness expression $S(\mathcal{V}, m)$, in terms of state variables $\mathcal{V}$ and menu m, SP *satisfies* $S(\mathcal{V}, m)$, denoted SP **sat** $S(\mathcal{V}, m)$, if

$$Inv_{SP} \wedge m \subseteq next_{SP} \wedge (\forall L \in \mathbf{L}.(L = \emptyset) \vee (m \cap L \neq \emptyset)) \Rightarrow S(\mathcal{V}, m)$$

In our context, liveness properties must be expressed as set expressions depending on the state variables and on a free variable m representing the possible menus of the process in any valid state.

For instance, an explicit liveness expression for the mutual exclusion example of figure 2 would be the following one:

$$\begin{aligned} S(\mathcal{V}, m) \stackrel{\text{def}}{=} & \ (\forall j \in U.st_j = thinking \Rightarrow req_j \in m) \\ \wedge & \ (\forall j \in U.st_j = eating \Rightarrow rel_j \in m) \\ \wedge & \ ((Nj \in U.st_j = hungry) > 0 \wedge \neg eaters \Rightarrow \\ & \ \exists k \in U.st_k = hungry \wedge ack_k \in m) \end{aligned}$$

Using invariant 4 and definition 13, the reader can prove that this property is satisfied by the arbiter of figure 2. From this property, weaker ones can be deduced. For instance, that if only one user k is *hungry* and no user is *eating*, then the event ack_k is mandatory for the system. Also, deadlock freedom expressed as $S(\mathcal{V}, m) \stackrel{\text{def}}{=} (m \neq \emptyset)$, can easily be proved.

4 Refinements

The *TCSP* refinement relation $\sqsubseteq$ preserves the safety and liveness properties of processes. Following the approach taken in TCSP, we say that a specification $SPEC$ is refined by another specification IMP if any possible trace of IMP is also allowed by $SPEC$, and if for every trace, any set of events that $SPEC$ is forced to offer, is also offered by IMP.

Definition 14 Given two specifications with the same alphabet, $SPEC$ and IMP, we say that IMP *is a refinement of* $SPEC$, denoted by $SPEC \sqsubseteq IMP$, if $[\![SPEC]\!] \sqsubseteq [\![IMP]\!]$.

The failures model enjoys a rich set of algebraic axioms to prove both equality and refinements of processes. For example, for any two processes P and P':

$$P \sqsubseteq P' \Leftrightarrow P \sqcap P' = P$$

The use of the algebraic laws to prove correctness of refinements has been shown elsewhere and will not be discussed here. The explicit modeling of process states by means of state variables allows us to compare guards and liveness rules in corresponding states i.e., in states reached after the same trace. To carry out this comparison, we establish some relation between states that takes the form of a predicate with free

variables of $\mathcal{V}_{sp}$ and $\mathcal{V}_{imp}$; this relation shall hold initially and be preserved by transitions. This proving technique has strong similarities with the concept of *bisimulation* introduced by Park [14] in the framework of *CCS*. The use of bisimulations to compare *TCSP* processes has been studied in other works [11].

In what follows, we assume as given two well-formed specifications with disjoint sets of state variables:

$$\begin{aligned} SPEC &= (\mathcal{L}_{spec}, \mathcal{V}_{spec}, \sigma_{0,spec}, \mathbf{TR}_{spec}, \mathbf{G}_{spec}, \mathbf{L}_{spec}) \\ IMP &= (\mathcal{L}_{imp}, \mathcal{V}_{imp}, \sigma_{0,imp}, \mathbf{TR}_{imp}, \mathbf{G}_{imp}, \mathbf{L}_{imp}) \end{aligned}$$

Definition 15 Given two well-formed specifications $SPEC$ and IMP, with $\mathcal{L} = \mathcal{L}_{spec} = \mathcal{L}_{imp}$, and predicates $Spec(\mathcal{V}_{spec})$ and $Imp(\mathcal{V}_{imp})$, which are $SPEC$-invariant and IMP-invariant respectively, we say that predicate ϕ, with free variables in $\mathcal{V}_{spec} \cup \mathcal{V}_{imp}$, is a *refinement relation* with respect to $SPEC$ and IMP if the following conditions hold:

1. $\overline{\sigma_0}(\phi) = true$ with $\sigma_0 = \sigma_{0,spec} \cup \sigma_{0,imp}$
2. $\forall e \in \mathcal{L}.Spec(\mathcal{V}_{spec}) \wedge Imp(\mathcal{V}_{imp}) \wedge G_e^{spec} \wedge G_e^{imp} \wedge \phi \Rightarrow TR_e(\phi)$
   with:
   $$TR_e = TR_e^{spec} \cup TR_e^{imp}$$

for any values of state variables in $\mathcal{V}_{spec}$ and $\mathcal{V}_{imp}$.

Proposition 16 Given two well-formed specifications $SPEC$ and IMP with $\mathcal{L} = \mathcal{L}_{spec} = \mathcal{L}_{imp}$,

$$SPEC \sqsubseteq IMP$$

if there are predicates $Spec(\mathcal{V}_{spec})$, $SPEC$-invariant, and $Imp(\mathcal{V}_{imp})$, IMP-invariant, and refinement relation ϕ with respect to $SPEC$ and IMP, satisfying the following conditions:

safety: $\forall e \in \mathcal{L}.Spec(\mathcal{V}_{spec}) \wedge Imp(\mathcal{V}_{imp}) \wedge \phi \wedge G_e^{imp} \Rightarrow G_e^{spec}$

liveness: $Spec(\mathcal{V}_{spec}) \wedge Imp(\mathcal{V}_{imp}) \wedge \phi \Rightarrow \mathbf{L}$
where:

$$\begin{aligned} \mathbf{L} \triangleq\ & \forall L(\mathcal{V}_{spec}) \in \mathbf{L}_{spec}.\ L(\mathcal{V}_{spec}) \neq \emptyset \Rightarrow \\ & \exists L'(\mathcal{V}_{imp}) \in \mathbf{L}_{imp}.L'(\mathcal{V}_{imp}) \neq \emptyset \wedge L'(\mathcal{V}_{imp}) \subseteq L(\mathcal{V}_{spec}) \end{aligned}$$

for any values of state variables in $\mathcal{V}_{spec}$ and $\mathcal{V}_{imp}$.

When refining some specification by another we define a more deterministic system, more amenable for practical implementation, possibly by adding and/or removing some state variables, while keeping unchanged the alphabet of events. If we are interested in designing distributed systems, we will eventually face the task of specifying a network of communicating subsystems, so that abstracting from internal activities, it becomes a refinement of the given system specification. Concealing a set of events in a *TCSP* process may produce divergence, if there is the possibility for infinite chattering to occur. The technique for proving divergence freedom is established below.

Definition 17 Given a process specification SP, with alphabet $\mathcal{L} \cup \mathcal{L}_h$, SP *is divergence free with respect to* $\mathcal{L}_h$ if the *TCSP* process $[\![SP]\!]\backslash\mathcal{L}_h$ is not divergent.

Proposition 18 Given a process specification SP, with alphabet $\mathcal{L} \cup \mathcal{L}_h$, SP *is divergence free with respect to* $\mathcal{L}_h$ if there exist a SP-invariant Inv, and a *variant* integer expression Ω with free variables in $\mathcal{V}$, satisfying the following conditions:

1. $Inv \Rightarrow \Omega \geq 0$
2. $\forall e \in \mathcal{L}_h . Inv \wedge G_e \Rightarrow TR_e(\Omega) < \Omega$

for any values of state variables in $\mathcal{V}$.

Definition 19 Given two well-formed specifications $SPEC$ and IMP with $\mathcal{L}_h = \mathcal{L}_{imp} - \mathcal{L}_{spec}$, IMP *is an implementation of* $SPEC$ *with respect to* $\mathcal{L}_h$ if:

1. IMP is divergence free with respect to $\mathcal{L}_h$
2. $SPEC \sqsubseteq IMP\backslash\mathcal{L}_h$

Definition 20 Given two well-formed specifications $SPEC$ and IMP, with $\mathcal{L}_h = \mathcal{L}_{imp} - \mathcal{L}_{spec}$, and given predicates $Spec(\mathcal{V}_{spec})$, $SPEC$-invariant, and $Imp(\mathcal{V}_{imp})$, IMP-invariant, the predicate ϕ with free variables in $\mathcal{V}_{spec} \cup \mathcal{V}_{imp}$, is an *abstraction relation* with respect to $SPEC$, IMP and $\mathcal{L}_h$if:

1. $\overline{\sigma}_0(\phi) = true$, with $\sigma_0 = \sigma_{0,spec} \cup \sigma_{0,imp}$
2. $\forall e \in \mathcal{L}_{spec} . Spec(\mathcal{V}_{spec}) \wedge Imp(\mathcal{V}_{imp}) \wedge G_e^{spec} \wedge G_e^{imp} \wedge \phi \Rightarrow TR_e(\phi)$ where:

$$TR_e = TR_{e,spec} \cup TR_{e,imp}$$

3. $\forall s \in \mathcal{L}_h . Spec(\mathcal{V}_{spec}) \wedge Imp(\mathcal{V}_{imp}) \wedge G_s^{imp} \wedge \phi \Rightarrow TR_s^{imp}(\phi)$

for any values of state variables in $\mathcal{V}_{spec}$ and $\mathcal{V}_{imp}$.

Proposition 21 Given well-formed specifications $SPEC$ and IMP, with $\mathcal{L}_h = \mathcal{L}_{imp} - \mathcal{L}_{spec}$, IMP is an implementation of $SPEC$ with respect to $\mathcal{L}_h$ if IMP is divergence free with respect to $\mathcal{L}_h$ and there are predicates $Spec(\mathcal{V}_{spec})$, $SPEC$-invariant, and $Imp(\mathcal{V}_{imp})$, IMP-invariant, and an abstraction relation ϕ with respect to $SPEC$, IMP and $\mathcal{L}_h$, satisfying the following conditions:

safety: $\forall e \in \mathcal{L}_{spec} . Spec(\mathcal{V}_{spec}) \wedge Imp(\mathcal{V}_{imp}) \wedge G_e^{imp} \wedge \phi \Rightarrow G_e^{spec}$

liveness: $Spec(\mathcal{V}_{spec}) \wedge Imp(\mathcal{V}_{imp}) \wedge Stable(\mathcal{V}_{imp}) \wedge \Phi \Rightarrow \mathbf{L}$
where $\mathbf{L}$ and $Stable$ are defined as

$$\mathbf{L} \triangleq \forall L \in \mathbf{L}_{spec} . (L \neq \emptyset \Rightarrow \exists L' \in \mathbf{L}_{imp} . L' \neq \emptyset \wedge L' - \mathcal{L}_h \subseteq L)$$
$$Stable \triangleq \forall L' \in \mathbf{L}_{imp} . (L' \neq \emptyset \Rightarrow L' - \mathcal{L}_h \neq \emptyset)$$

for any values of state variables in $\mathcal{V}_{spec}$ and $\mathcal{V}_{imp}$.

5 Parallel composition of process specifications

Once we have shown that the desired network exhibits the intended behaviour, we are in position to determine the specifications for the component subsystems. For every subsystem, the design process is resumed and new decisions are taken to define new refinements and, possibly, new implementations. Eventually, the whole system is constructed in a hierarchical way. To achieve this goal, parallel composition of specifications with state variables must be defined.

If we restrict our attention to deterministic processes, this is done in a straightforward way. In deterministic processes, liveness rules may be omitted since they are given by the guards. Also, state variables in every subsystem either evolve according to their own transition rules, or remain unchanged when the event is performed solely by the other subsystem. If they synchronize in one event, the guard of that event is obtained by the conjunction of the guards for that event in both subsystems. Otherwise, the guard is imported from the corresponding subsystem. These ideas are reflected in the following definition:

Definition 22 Given two deterministic specifications,

$$\begin{aligned} SP_1 &= (\mathcal{L}_1, \mathcal{V}_1, \sigma_{0_1}, \mathbf{TR}_1, \mathbf{G}_1) \\ SP_2 &= (\mathcal{L}_2, \mathcal{V}_2, \sigma_{0_2}, \mathbf{TR}_2, \mathbf{G}_2) \end{aligned}$$

with disjoint sets of state variables, *the parallel composition of* SP_1 *and* SP_2, denoted by $SP_1 \| SP_2$, is given by the tuple, $(\mathcal{L}, \mathcal{V}, \sigma_0, \mathbf{TR}, \mathbf{G})$ where:

1. $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$
2. $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$
3. $\sigma_0 = \sigma_{0_1} \cup \sigma_{0_2}$
4. $\mathbf{TR}$ is the family of transition rules indexed by $\mathcal{L} \times \mathcal{V}$ and defined by:

$$E_{e,x} \triangleq \begin{cases} E^1_{e,x} & \text{if } e \in \mathcal{L}_1 \wedge x \in \mathcal{V}_1 \\ E^2_{e,x} & \text{if } e \in \mathcal{L}_2 \wedge x \in \mathcal{V}_2 \\ x & \text{if } (e \notin \mathcal{L}_1 \wedge x \in \mathcal{V}_1) \vee (e \notin \mathcal{L}_2 \wedge x \in \mathcal{V}_2) \end{cases}$$

5. $\mathbf{G}$ is the family of deterministic guards indexed by $\mathcal{L}$ and defined by:

$$G(e) \triangleq \begin{cases} G_1(e) & \text{if } e \in \mathcal{L}_1 - \mathcal{L}_2 \\ G_2(e) & \text{if } e \in \mathcal{L}_2 - \mathcal{L}_1 \\ G_1(e) \wedge G_2(e) & \text{if } e \in \mathcal{L}_1 \cap \mathcal{L}_2 \end{cases}$$

This tuple actually defines a deterministic specification.

Fact 23 Given two deterministic specifications SP_1 and SP_2, then

$$[\![SP_1 \| SP_2]\!] = [\![SP_1]\!] \| [\![SP_2]\!]$$

Using this definition we are in position to analyze some interesting parallel and distributed algorithms but it is worthwhile extending the above ideas to cover the most general situation, i.e. the composition of non-deterministic systems.

Unfortunately, the definition of the parallel composition of nondeterministic process specifications is rather cumbersome and not very useful. The authors have found more interesting in practice to have criteria to *decompose* a given specification, representing a complex system, into a number of smaller specifications in a such way that, composing in parallel these specifications, we get a behaviour identical to that of the complex system. The details cannot be given here due to lack of space. They can be found in [1].

6 Conclusions

A technique for the specification and verification of *TCSP* processes has been presented, establishing a method with a sound mathematical basis for the hierarchical development of communicating systems. The authors have extensively tried the method in a number of examples that cover distributed arbitration systems and other synchronization problems, like the dinning and drinking philosophers, or distributed message routing systems. For more details, interested readers are addressed again to [1].

References

[1] L.M. Alonso. *Técnicas formales para el desarrollo jerárquico de sistemas concurentes.* PhD thesis, Universidad del País Vasco, 1992.

[2] L.M. Alonso and R. Peña. Acceptance automata: a framework for specifying and verifying TCSP parallel systems. In E.H.L. Aarts, J. van Leeuwen, and M. Rem, editors, *PARLE'91: Parallel Architectures and Languages Europe*, pages 75–91, Springer-Verlag, 1991.

[3] L.M. Alonso and R. Peña. Using state variables for the specification and verification of TCSP processes. *Internal report DIA-UCM-92.3, Dep. Informática y Automática, Univ. Complutense de Madrid, Spain*, 1–26, 1992.

[4] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[5] S.D. Brookes. *A Model for Communicating Sequential Processes.* PhD thesis, Oxford University, 1983.

[6] S.D. Brookes, A.W. Roscoe, and C.A.R. Hoare. A theory for communicating sequential processes. *Journal of the ACM*, 31:560–599, 1984.

[7] M. Broy and M. Wirsing. Partial abstract types. *Acta Informatica*, 18:47–64, 1982.

[8] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, London, 1989.

[9] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, London, 1985.

[10] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[11] H. Jifeng. Process simulation and refinement. *Formal Aspects of Computing*, 1:229–241, 1989.

[12] R. Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1980.

[13] R. Milner. *Communication and Concurrency.* Prentice-Hall, London, 1989.

[14] D. Park. Concurrency and automata on infinite sequences. In E.H.L. Aarts, J. van Leeuwen, and M. Rem, editors, *Proceedings 5th GI Conf. of Theoretical Computer Science*, pages 245–251, Springer-Verlag, 1981. Lecture Notes in Computer Science.

[15] R. Peña and L.M. Alonso. Specification and Verification of TCSP Systems by Means of Partial abstract Types. In J. Diaz; F. Orejas, editor, *TAPSOFT'89: Theory and Practice of Software Development, Vol. 2*, pages 328–344, Springer-Verlag, 1989. Lecture Notes in Computer Science no. 352.