REAL-TIME PROGRAMMING AND ASYNCHRONOUS MESSAGE PASSING

Ron Koymans

Jan Vytopil

Willem P. de Roever

RUU-CS-83-9

June 1983

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–53 1454
The Netherlands

REAL-TIME PROGRAMMING AND ASYNCHRONOUS MESSAGE PASSING

Ron Koymans

Jan Vytopil

Willem P. de Roever

Technical Report RUU-CS-83-9

June 1983

Department of Computer Science

University of Utrecht

P.O. Box 80.002

3508 TA Utrecht

the Netherlands

REAL-TIME PROGRAMMING AND ASYNCHRONOUS MESSAGE PASSING *)

Ron Koymans
Department of Computer Science
University of Utrecht, the Netherlands

Jan Vytopil
Philips' Telecommunicatie Industrie B.V.
PTI Hilversum, the Netherlands

Willem P. de Roever
Department of Computer Science
University of Utrecht, the Netherlands
and
Faculty of Science
University of Nijmegen, the Netherlands

Abstract. This paper indicates a method of describing real-time processes and their asynchronous communication by means of message exchanges. This description method is based upon an extension of linear time temporal logic to a special temporal logic in which real-time and asynchronous message passing properties can be expressed. We give a model of this logic, define new operators and show amongst others how they can be applied to specify real-time asynchronous message passing and an abstract real-time transmission medium.

*)To appear in: Proceedings of the 2nd Conference on Principles of Distributed Computing, August 17-19, Montreal, Canada, 1983.

§1. Introduction

1. Motivation

In recent years an acute need has arisen for a methodology of real-time processes, as they occur in telecommunication systems or process control systems in general. This paper is an attempt to contribute to the development of such a methodology.

The paper concerns the description of real-time processes and their asynchronous communication by means of signal (message) exchanges. For the characterization of these processes and their communication we develop a special temporal logic with two new operators. By means of these we believe we are able to express the real-time and asynchronous communication aspects in a natural way. As a consequence, by using a proof system for our temporal logic, one can reason about real-time processes and their communication, i.e. one can prove that a program meets its real-time specifications.

Of course, a complete real-time methodology is still some time off, but we believe that our method can be applied to real-time processes in general thus providing a possible direction in which such a methodology can develop.

We illustrate our study of real-time processes and their asynchronous communication by means of a very small CHILL-like language in which the essence of asynchronous real-time communication is present. CHILL (CCITT HIgh Level Language, see [CHILL]) is the recommended language of the CCITT (Comité Consultatif Internationale de Télégraphique et Téléphonique). CHILL is an interesting subject of study as it is used by many different PTT's and industries, such as Siemens, NTT(Nippon Telephone and Telegraph), ITT(International Telephone and Telegraph) and PTI(Philips' Telecommunicatie Industrie B.V.) for programming telecommunication systems.

In the area of distributed processing recent research has focussed on languages such as CSP (see [HOARE]) and ADA (see [ADA]) for which safety and liveness properties etc. have been extensively studied (see e.g. [KUIPER & DE ROEVER], [MANNA & PNUELI], [PNUELI & DE ROEVER], [GERTH & DE ROEVER]). These languages, however, are based on the rendezvous mechanism which requires synchronous communication. Furthermore, the research has until now almost completely ignored the real-time aspect although a suggestion has been made for the ADA-case at the end of [PNUELI & DE ROEVER] and a first step towards a proof technique for real-time properties of concurrent programs has been made in [BERNSTEIN & HARTER].

2. Problems faced and solutions suggested

We take the following real-time aspects into account:

A. a time-out mechanism:
in some present day concurrent programming languages a timer can be present for the purpose of restricting the period of waiting for a message to some fixed amount of time. Usual temporal logics cannot express this as they consider time qualitatively and not quantitatively. The need for a time-out possibility is obvious when developing telecommunication systems as in a certain amount of time there must always be some progression in the system.

B. a medium real-time property:
we impose the condition that the transmission medium is not "too lazy", i.e. the medium may not stand idle for too long if there are outstanding signals to be transmitted (again this is an obvious requirement in a telecommunication environment); the formal expression of this property is given in §3.

The following technical complication concerning aspect (A) arises: if program control reaches (an occurrence of) an action (this is the CHILL-analog of a statement) which can receive a signal (from now on called receive case action), it must be stated that a timer will expire after a specified time. We take care of this by introducing a special "timerstart"-location (the notion of location is explained and refined below) for each receive case action. Similar cases of introducing "locations" which do not actually occur in a program in order to reason (prove properties) about that program arise for Brinch Hansen's DP (see [BRINCH HANSEN], [ASTESIANO & ZUCCA]) and ADA (see [GERTH & DE ROEVER]).

We now refine the notion of location in a program. This is needed to be able to recognize the difference between occurrences of actions. In Pnueli's (and also Lamport's) works location predicates are introduced, i.e. predicates true in case control resides at a specific location of a program. For the simple kind of programs that they consider these predicates are characterized syntactically (e.g. by pointing to a place in the program text). Once (dynamic) process creation is involved this simple syntactic characterization is not sufficient anymore: one can still point to a specific place in the text of a process definition but one can not tell the difference between this textual place in one instance of the process definition and the same textual place in another instance of the process definition. To identify uniquely every action of sending or receiving, e.g. as is required in the expression of "every signal received has been sent before", we need a third component (apart from syntactic location, i.e. textual place, and process instance). This third component contains the number of times this specific syntactic location in this process instance has been passed (this is needed because, for example, a send action can textually be placed in a loop and hence it can be passed several times).

Apart from the real-time property B above, we impose in our formulation the following constraints on the transmission medium: it may not create ghost messages (either original or copies of real signals) and it must be fair relative to the transmission of signals (see below). Furthermore, the medium need not be error-free (but fairness, see below, guarantees that it can not be too faulty either). Summarizing, the medium has the following properties:
- it is not too lazy,
- it may not create ghost messages,
- it is fair relative to the transmission of signals,
- it may loose messages (we assume that corrupted messages are discarded at a lower level),
- it may reorder messages.

Fairness of the medium must be imposed, as the properties described hitherto do not guarantee that anything will be received. The fairness (liveness) condition imposed on the medium then is classical: if infinitely often a signal from a collection of signals is sent, then at least one signal from the collection will arrive at its destination. See §3 for some consequences of this fairness condition. In some programming languages (such as CHILL) it is possible that more than one signal is receivable at a time (cf. the accept-statement in ADA). In that case one might impose another fairness condition, namely fairness relative to the receipt of signals: if more than one signal is receivable, we demand that the choice mechanism is fair to all receivable signals (in ADA a weaker fairness condition, namely that the choice mechanism given a specific entry is fair, is explicitly defined in the language by entry-queues but in [PNUELI & DE ROEVER] it was shown how to avoid defining fairness by such implementation oriented concepts). These two fairness conditions are illustrated by a programming example in §5.

In order to describe real-time and asynchronous message passing properties we use a specialized temporal logic which is obtained by extending linear time temporal logic (see [LAMPORT]) with the operator $\diamondminus$ (before), referring to the past and real-time operators such as $\Diamond_{\leq t}$ (eventually within real-time t from now) and $\Box_{>t}$ (always after real-time t has elapsed from now). In this paper we call this specialized temporal logic real-time temporal logic. We derive all our real-time operators from the real-time operator $\mathcal{U}_{=t}$ (strong until in real-time t). Together with the before operator $\diamondminus$ this demands a new temporal logic model. This model has states as in usual temporal logic models but now a state has a real-time component conceptually representing a kind of global clock. Therefore a necessary condition for a state transition is that time cannot decrease (we allow a state transition to leave this real-time component the same: if program control is at the beginning of a receive case action we stipulate that program control transfers immediately, i.e. without any real-time elapsing, to the above mentioned "timerstart"-location, see §4). The before operator $\diamondminus$ (which is new in the context of program proving) is used for the formulation of the real-time property of the medium (see §3) and furthermore for the classical property that a received signal must have been previously sent (see §3). In the formulation of the latter property we already recognize the natural way we can express this property by an operator which refers to the past.

3. Related work

The only related work concerning a proof technique for real-time properties of concurrent programs that we know of is [BERNSTEIN & HARTER]. Because real-time is involved their communication is of asynchronous nature. However, their work differs from ours in several aspects. Firstly, they do not consider dynamic process creation. Secondly, they consider only real-time operators related to the temporal implication operator while we introduce a general real-time operator $\mathcal{U}_{=t}$ from which many more (much needed!) real-time operators can be derived. Thirdly, their model for program execution is more complicated than ours due to restrictions concerning interleaved execution and indivisibility of operations. Also their modelling of execution time does not in any way correspond to the execution time in reality while our modelling of execution time can be directly related to real-time execution. In general, a multiprogramming model contrasts with the practice of real-time programming in which independent execution on separate processors seems to be the rule once real-time considerations become critical. Fourthly, their work concentrates on safety properties while ours is concerned mostly with liveness properties since these are what real-time programming is about.

Related work concerning asynchronous message passing on its own is more readily available. We mention [MISRA, CHANDY & SMITH] describing safety and liveness properties of message passing networks by a hierarchical method based upon combining the specifications of component processes to obtain the specification of the network and [SCHLICHTING & SCHNEIDER] concerning inference rules for proving partial correctness of concurrent programs that use message passing for synchronisation and communication.

§2. The logic and its model

Our model is different from the usual temporal logic model, because real-time must be considered. The model still is ultimately state-based (as for usual temporal logics) but a state now has a real-time component. This real-time component will be an element of the real-time domain $\mathfrak{R}$:

> $\mathfrak{R}$ is a structure $(|\mathfrak{R}|, <, +)$ with a linear ordering $<$ and addition $+$ and the following constraint:
> $\mathfrak{R}$ is a superstructure (an extension) of $(\omega, <, +)$ where ω is the set of natural numbers (including 0) with its usual ordering and addition.

Remark: $\mathfrak{R}$ can be specified to suit certain applications, e.g. an element ∞ could be added which denotes an infinite waiting time.

Notation: Let $\leq$ be the usual analog of $<$.
By $|\mathfrak{R}|_{\geq 0}$ we denote $\{ t \in |\mathfrak{R}| \mid 0 \leq t \}$.

For our logic we assume a many-sorted language with the standard operators $\perp$ (falsity), $\rightarrow$ (implication) and $\forall$ (universal quantifier, for all sorts) and the temporal operators $\mathcal{U}_{=t}$ (strong until in real-time t) and $\Diamond$ (before). We demand that $|\mathfrak{R}|_{\geq 0}$ is one of the sorts of the language. We assume $=$ (equality) to be part of the language.

The sentences of the language are formed by means of these operators and atomic sentences which denote state predicates. As usual in temporal logic models, we have special state predicates, called location predicates, of the form *at*(l) and *after*(l) where l is a location (note that we refer here to the refined notion of location, see §1.2). We will call these location predicates control points; *at*(l) is true either when program control is just before the action belonging to l or when l is a timerstart-location (as introduced in §1.2) and program control is at this timerstart-location; *after*(l) is true either when program control is just after the action belonging to l or when l is a timerstart-location and program control is just after this location (a full discussion of location predicates requires an operational semantics in the style of [PNUELI & DE ROEVER]). By our refinement of the notion of location a program can only be once consecutively at and be once consecutively after a location.

In our model each state has exactly one control point for every process. A state s is a quadruple

$$< t, PI, \{ C_i \mid i \in PI\}, PS >$$

where

t ∈ |ℜ|,
PI is the set of process instances,
C_i is the control point for process instance i
and
PS is the program status, i.e. the set of all values of all program variables.

Note that the real-time domain ℜ occurs in both the logic and its model; in fact, we have a new logic and its accompanying model for every specification of ℜ to a concrete structure.

In our model, there is a unique initial state I. This initial state has real-time component 0.

A computation is considered as an infinite sequence $< s_i >_{i=0}^{\infty}$ where s_i is a state for all $i \in \omega$, $s_0 = I$ and $\pi_1(s_i) \leq \pi_1(s_{i+1})$ for all $i \in \omega$, where π_j is the projection function on the j-th coordinate (i.e. time cannot decrease). We demand, furthermore, that one of the following cases applies:

— $s_i \neq s_j$ for all $i, j \in \omega$, $i \neq j$
— there is a $i_0 \in \omega$ such that $s_{i_0+n} = s_{i_0}$ for all $n \in \omega$ and $s_i \neq s_j$ for all $i, j \in \omega$, $0 \leq i < j \leq i_0$
(this case corresponds to the case of a terminating computation where s_{i_0} is the end-state of the computation; since this end-state does not change anymore as time elapses, it is repeated for ever).

Let $\sigma = < s_i >_{i=0}^{\infty}$ be a computation. For $i \in \omega$ define σ_i to be s_i. In this paper $\Rightarrow$ denotes implication in the meta-language. Quantification in the meta-language is not indicated by special symbols because it is always clear from the context whether quantification in the language itself or in the meta-language is meant.

We now inductively define the relation $(\sigma, j) \models \phi$ for all pairs (σ, j) (where σ is a computation and $j \in \omega$) and any real-time temporal logic formula ϕ as follows:

$(\sigma, j) \models P \overset{def.}{:=} P(\sigma_j)$ for P a state predicate

$(\sigma, j) \not\models \bot$

$(\sigma, j) \models P \rightarrow Q \overset{def.}{:=} (\sigma, j) \models P \Rightarrow (\sigma, j) \models Q$

$(\sigma, j) \models \forall v \in D \ P(v) \overset{def.}{:=} \forall v \in D \ ((\sigma, j) \models P(\underline{v}))$ for all sorts D

$$(\sigma, j) \models P\, \mathcal{U}_{=t}\, Q \overset{def.}{:=} \exists n \in \omega\, [\sigma_{j+n} \models Q \wedge \pi_1(\sigma_{j+n}) = \pi_1(\sigma_j) + t$$
$$\wedge \forall i \in \omega\, [0 \leq i < n \rightarrow \sigma_{j+i} \models P]]$$

$(\sigma, j) \models \blacklozenge P \overset{def.}{:=} \exists n \in \omega\, [0 \leq n \leq j \wedge \sigma_{j-n} \models P]$.

In our logic the operator $\mathcal{U}_{=t}$ is considered as a ternary operator with formulae as its first two arguments and an element from $\Re_{\geq 0}$ as its third argument.

The operator $\mathcal{U}_{=t}$ is needed for the expression of real-time properties, while $\blacklozenge$ leads to the elegant expression of properties which refer to the past, e.g. when a signal is received, it was once sent.

We define furthermore the following derived operators for formulae P and Q:

$\neg P \overset{def.}{=} P \rightarrow \bot$ (negation)

$P \wedge Q \overset{def.}{=} \neg(P \rightarrow \neg Q)$ (conjunction)

$P \vee Q \overset{def.}{=} \neg P \rightarrow Q$ (disjunction)

$P \leftrightarrow Q \overset{def.}{=} (P \rightarrow Q) \wedge (Q \rightarrow P)$ (equivalence)

$\exists x \in \mathcal{D}\ P(x) \overset{def.}{=} \neg \forall x \in \mathcal{D}\ \neg P(x)$ for all sorts $\mathcal{D}$ (existential quantification)

$\top \overset{def.}{=} \neg \bot$ (truth)

$\diamond_{=t} P \overset{def.}{=} \top\, \mathcal{U}_{=t} P$ (eventually in real-time t)

$\diamond P \overset{def.}{=} \exists t \in \Re_{\geq 0}\ \diamond_{=t} P$ (eventually)

$\diamond_{<t} P \overset{def.}{=} \exists t' \in \Re_{\geq 0}\ (t' < t \wedge \diamond_{=t'} P)$ (eventually before real-time t)

$\diamond_{>t} P \overset{def.}{=} \exists t' \in \Re_{\geq 0}\ (t < t' \wedge \diamond_{=t'} P)$ (eventually after real-time t)

$\diamond_{\leq t} P \overset{def.}{=} \diamond_{=t} P \vee \diamond_{<t} P$ (eventually within real-time t)

$\diamond_{\geq t} P \overset{def.}{=} \diamond_{=t} P \vee \diamond_{>t} P$ (eventually from real-time t)

$\Box P \overset{def.}{=} \neg \diamond \neg P$ (henceforth)

$\Box_{\mathcal{R} t} P \overset{def.}{=} \neg \diamond_{\mathcal{R} t} \neg P$ for $\mathcal{R} \in \{ =, <, >, \leq, \geq \}$ (henceforth in/before/after/within/from real-time t)

$P\, \mathcal{U}\, Q \overset{def.}{=} \exists t \in \Re_{\geq 0}\ (P\, \mathcal{U}_{=t}\, Q)$ (strong until)

$P\, \tilde{\mathcal{U}}\, Q \overset{def.}{=} \Box P \vee P\, \mathcal{U}\, Q$ (weak until)

Concerning precedence of operators we give one-place operators (such as $\blacklozenge, \neg$) highest precedence. The operators $\mathcal{U}_{=t}$, $\mathcal{U}$ and $\tilde{\mathcal{U}}$ have higher precedence than $\wedge$ and $\vee$ which, in their turn, have higher precedence than $\rightarrow$ and $\leftrightarrow$.

The defined operators $\Diamond$, $\Box$, $\mathcal{U}$ and $\tilde{\mathcal{U}}$ above correspond indeed to the well-known operators of the same name. As an example of derivable general formulae, we give here the interactions of the new operator $\blacklozenge$ with the four old ones (see Appendix B for an informal derivation of these formulae and the interaction of $\blacklozenge$ with other operators):

1. $\blacklozenge \Diamond P \leftrightarrow \Diamond \blacklozenge P \leftrightarrow \blacklozenge P \vee \Diamond P$

2. $\blacklozenge \Box P \leftrightarrow \Box P$ and $\Box \blacklozenge P \leftrightarrow \blacklozenge P$

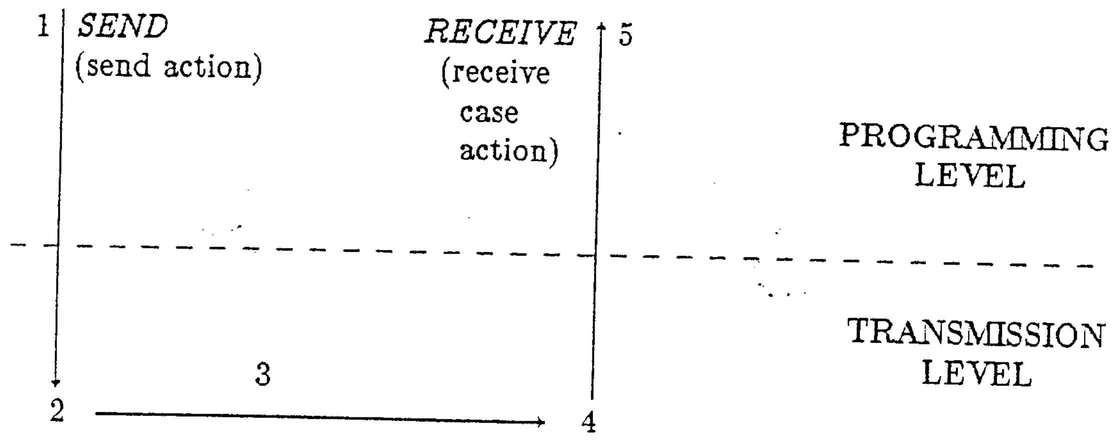
3. $\blacklozenge (P \, \mathcal{U} \, Q) \rightarrow \blacklozenge P \; \mathcal{U} \; \blacklozenge Q$ (the reverse implication is false)

4. $\blacklozenge (P \, \tilde{\mathcal{U}} \, Q) \rightarrow \blacklozenge P \; \tilde{\mathcal{U}} \; \blacklozenge Q$ (the reverse implication is false)

For non-temporal operators, $\blacklozenge$ interacts the same as $\Diamond$ ($\blacklozenge$ is the "dual" of $\Diamond$ in time but for the rest they correspond).

§3. An abstract transmission medium

In the following we give an abstract view of asynchronously communicating processes. We assume the only means of communication to be signals. Signals are sent by a sending process (the sender) and received by a receiving process (the receiver). In general there are 5 consecutive (chronologically seen) phases for each signal depicted as follows:



1. the signal is sent (the send action takes place),
2. the signal is waiting for transmission,
3. the signal is being transmitted,
4. the signal has arrived at its destination and is waiting for receipt,
5. the signal is received.

Note that we view phase 1 and 5 as instantaneous actions, e.g. phase 5 describes the moment the signal is actually accepted by a receive case action (program control, however, could have already been at the receive case action much longer).

Because we describe asynchronous communication, the sender simply moves on after a send action, i.e. after phase 1 the sender will not wait for phases 2 to 5 to finish.

As already stated in §1.2, we describe a transmission medium with the following properties: not too lazy, no creation of ghost messages, fair relative to the transmission of signals, possibly losing messages and possibly reordering messages.

We now illustrate how the properties of the medium can be expressed in our logic. The most interesting case is the medium real-time property (the medium is not too lazy):

$$
\begin{aligned}
&\exists c \in \mathbb{R}_{\geq 0}\, \Box \forall s_1 \in \mathcal{SEND} \\
&\quad [[\blacklozenge\, \mathit{after}(\mathit{location}(s_1)) \wedge \neg \blacklozenge\, \mathit{arrived}(s_1)] \\
&\quad \rightarrow [\exists s_2 \in \mathcal{SEND}\ [\neg \blacklozenge\, \mathit{arrived}(s_2) \wedge \Diamond_{\leq c}\, \mathit{arrived}(s_2)] \\
&\qquad \vee \Box_{> c}\, \neg \mathit{arrived}(s_1)]].
\end{aligned}
$$

In this axiom $\mathcal{SEND}$ is a semantic domain (as in denotational semantics) corresponding to the send action of the programming language under consideration. In general, $\mathcal{SEND}$ consists of several components including for example a signal identification and a destination (a process instance). We demand that every $s \in \mathcal{SEND}$ has a location component, denoted by *location*(s) which identifies the location belonging to the send action s (remember that a location has three components: textual place in the program text, process instance and number of times this action already has been executed at this place and in this instance - see §1.2 -); *arrived* is a predicate which corresponds to signal transmission phase 4: *arrived* indicates for a send action s whether the signal corresponding to s has arrived at its destination. This axiom reads as follows: there is a non-negative real-time constant c such that whenever there is a signal that has not yet arrived from a send action s_1 then *either* <u>some</u> signal will arrive within real-time c *or* after real-time c the signal from s_1 will never arrive. Note that c is an implementation dependent constant and can be related to the maximum transmission time of the medium in the real world.

A straightforward consequence of the axiom above is: there is a real-time constant c such that the arrival of a sent, but not yet arrived, signal can not be arbitrarily long delayed, unless in every period of real-time c *some* signal arrives. The expression of this consequence in our logic is as follows:

$$
\begin{aligned}
&\exists c \in |\Re|_{\geq 0}\ \exists d \in |\Re|_{\geq 0}\ \Box \forall s_1 \in \mathcal{SEND} \\
&\quad [[\blacklozenge\, \mathit{after}(\mathit{location}(s_1)) \wedge \neg \blacklozenge\, \mathit{arrived}(s_1) \\
&\quad\ \wedge \neg \exists s_2 \in \mathcal{SEND}\ [\neg \blacklozenge\, \mathit{arrived}(s_2) \wedge \Diamond_{\leq c}\, \mathit{arrived}(s_2)]] \\
&\quad \rightarrow [\Diamond_{\leq d}\, \mathit{arrived}(s_1) \vee \Box \neg\, \mathit{arrived}(s_1)]].
\end{aligned}
$$

This consequence can be informally deduced as follows: take c and d equal to the c from the axiom above and suppose
$\blacklozenge\, \mathit{after}(\mathit{location}(s_1)) \wedge \neg \blacklozenge\, \mathit{arrived}(s_1)$
$\wedge \neg \exists s_2 \in \mathcal{SEND}\ [\neg \blacklozenge\, \mathit{arrived}(s_2) \wedge \Diamond_{\leq c}\, \mathit{arrived}(s_2)]$
for an arbitrary program state and an arbitrary $s_1 \in \mathcal{SEND}$.
According to the above axiom we then have $\Box_{>c} \neg\, \mathit{arrived}(s_1)$.
Suppose also $\Box_{\leq c} \neg\, \mathit{arrived}(s_1)$ is valid.
In that case $\Box_{>c} \neg\, \mathit{arrived}(s_1) \wedge \Box_{\leq c} \neg\, \mathit{arrived}(s_1)$ holds and this implies $\Box \neg\, \mathit{arrived}(s_1)$ (This implication and many others are easily derived from the definition of the operators in §2.)
In the other case $\neg \Box_{\leq c} \neg\, \mathit{arrived}(s_1)$ is valid.
But from the definition of $\Box_{\leq c}$ this is the same as $\Diamond_{\leq c}\, \mathit{arrived}(s_1)$ and as d=c it thus follows $\Diamond_{\leq d}\, \mathit{arrived}(s_1)$.

The expression in our logic that the medium may not create ghost messages is fairly simple due to the presence of the before operator: a condition (among others, see §4) for a signal to be received by a receive case action is that it must have previously been sent. In our logic this condition can be formulated as $\blacklozenge\, \mathit{after}(\mathit{location}(\text{send}))$ where $\text{send} \in \mathcal{SEND}$ is the send action corresponding to the signal.

The expression of the fairness property of the medium is classical:

$\Box\forall$senders $\subseteq$ $\mathcal{SEND}$
  [$\Box\Diamond\exists$send $\in$ senders [*after*(*location*(send))]
  $\rightarrow\Diamond\exists$send $\in$ senders [*arrived*(send)]].

It just states that whenever infinitely often a signal from the collection senders is sent, then at least one of these signals will arrive. This implies by classical temporal logic reasoning that even infinitely many of these signals will arrive. In this case this fairness condition has an even stronger consequence as the following corollary shows.

Corollary: Only a finite number of signals can get lost.
(Peter van Emde Boas)

Proof: Suppose infinitely many signals get lost. Take these infinitely many signals as the collection senders in the fairness condition above. As all these signals have been sent (since they get lost), infinitely often a signal from this collection has been sent. The conclusion of the fairness condition above is that at least one signal from the collection arrives. A contradiction.

Remark: Theoretically, this corollary is more or less unwanted. Such a strong corollary is possible because of the fact that there is no restriction on the collection of signals in the fairness condition. A theoretical alternative would be to restrict this collection, e.g. by demanding that the signals come from one and the same process instance. Such solutions, however, appear not to be very convincing in the general case. We can afford to have such a strong fairness condition as in practice the difference is immaterial: one can give no upperbound on the number of lost signals (in other words: one cannot use in any way the fact that from some moment on signals will not get lost anymore). One must realize that this problem arises because we intend to axiomatize the underlying datastructure (c.q. the medium) on its own, instead of deriving the axiomatization syntactically by means of some scheme for all programs which use this datastructure. This problem deserves further investigation.

The remaining two properties of the medium, namely that it may loose messages and it may reorder messages need no explicit formulation in our logic since the absence of any conditions concerning these properties implicitly acknowledges their possibility.

§4. Real-time asynchronous communication properties

In §3 we already described (real-time) communication properties of the medium. Now we concentrate on the communication properties of the processes. In the following we consider a programming language P with at least the following properties: in P it is possible to declare and dynamically start processes which can communicate by means of signals; the communication is asynchronous and signals can be sent by a send action and received by a receive case action (we also assume that a receive case action has an optional time-out mechanism, see §1.2 real-time aspect A).

In general, we need only the following axiom when describing a send action in an asynchronous communication environment:

$$\Box \forall \mathrm{send} \in \mathcal{SEND}\ [\, at(location(\mathrm{send})) \\ \rightarrow at(location(\mathrm{send}))\ \mathcal{U}\ after(location(\mathrm{send}))].$$

Note that this axiom expresses only that a send action terminates within a finite amount of real-time (and that is exactly the asynchronous aspect of the send action: there is no rendezvous mechanism as in CSP or ADA).

For a receive case action the situation is far more complex.

Because we consider the possibility of a timed receive case action, we have to introduce a timer and a "timerstart"-location for each receive case action. We can describe by the following axiom that the timer expires *exactly at a certain time after* the arrival at the receive case action no matter whether a signal does or does not arrive in the meantime:

$$
\begin{aligned}
&\Box\forall \text{rec} \in \mathcal{RECEIVE} \\
&\quad [at(location(\text{rec})) \\
&\quad \rightarrow [at(location(\text{rec}))\, \mathcal{U}_{=0}\, at(timerstartlocation(\text{rec})) \\
&\qquad \wedge \Diamond_{=timevalue(\text{rec})}[timer(\text{rec}) = 0] \\
&\qquad \wedge \neg\Diamond_{<timevalue(\text{rec})}[timer(\text{rec}) = 0]]].
\end{aligned}
$$

In this axiom $\mathcal{RECEIVE}$ is the semantic domain corresponding to the receive case action of P. Similar remarks about the structure of the domain $\mathcal{RECEIVE}$ apply as in the case of the domain $\mathcal{SEND}$ (see §3); *location*(rec) has an analogous meaning as for send $\in \mathcal{SEND}$; *timerstartlocation*(rec) denotes the associated timerstart-location of rec and *timer* the associated timer of rec; *timevalue*(rec) denotes the maximum real-time during which control waits at *timerstartlocation*(rec) for a signal to arrive. This axiom expresses the following: when program control transfers to a receive case action then the associated timer expires at the associated timevalue and not before. As the timer expires exactly at the associated timevalue, it is conceptually impossible (when considering the axiom above) to let in the meantime the control point remain at the beginning of the receive case action (this would be selfcontradictory as the control point at the beginning of the receive case action identifies the beginning of the real-time interval in question and this beginning must be a unique fixed reference point relative to which the timer runs). So the timerstart-location had to be introduced and the timerstart-location control point had to be reached with no real-time elapsing. This accounts for the clause $at(location(\text{rec}))\, \mathcal{U}_{=0}\, at(timerstartlocation(\text{rec}))$ in the axiom above.

In general one needs at least two more axioms for a receive case action to describe the following:

1. the conditions which cause progress when program control is at a receive case action

2. the action undertaken when program control is at a receive case action and a signal arrives.

In the special case that the semantics of the receive case action of P is simply to wait for a signal to arrive with an optional time-out mechanism specified in an else part to be executed when the time-out occurs, we can formulate the following axioms to describe 1 and 2 above:

$$\Box\forall \text{rec} \in \mathcal{RECEIVE}\ [\,at(timerstartlocation(\text{rec})) \rightarrow at(timerstartlocation(\text{rec}))\ \tilde{\mathcal{U}}\ [timer(\text{rec}) = 0 \vee \exists \text{send} \in \mathcal{SEND}\ receivable(\text{send}, \text{rec})]]$$

$$\Box\forall \text{rec} \in \mathcal{RECEIVE}\ [[\,at(timerstartlocation(\text{rec})) \wedge timer(\text{rec}) = 0 \wedge \neg\exists \text{send} \in \mathcal{SEND}\ receivable(\text{send}, \text{rec})] \rightarrow at(timerstartlocation(\text{rec}))\ \mathcal{U}_{=0}\ at(else(\text{rec}))]$$

$$\Box\forall \text{rec} \in \mathcal{RECEIVE}\ [[\,at(timerstartlocation(\text{rec})) \wedge \exists \text{send} \in \mathcal{SEND}\ receivable(\text{send}, \text{rec})] \rightarrow \exists \text{send} \in \mathcal{SEND}\ [[at(timerstartlocation(\text{rec})) \wedge receivable(\text{send}, \text{rec})]\ \mathcal{U}_{=0}\ [at(chosenalternative(\text{send}, \text{rec})) \wedge \Box\neg\, arrived(\text{send})]]].$$

The predicate *receivable*(send, rec) in the axioms above indicates the conditions for a signal corresponding to send $\in \mathcal{SEND}$ to be received by the receive case action rec $\in \mathcal{RECEIVE}$. In §3 we already stated that one of these conditions is for example $\diamondsuit\!\!\!\!\!\!\longleftrightarrow$ *after*(*location*(send)) which excludes the creation of ghost messages. A further condition is of course *arrived*(send) (see §3 for the predicate *arrived*). Other conditions are strongly dependent on P. In the second axiom, *else*(rec) denotes the location belonging to the else part of the receive case action. In the third axiom, *chosenalternative*(send, rec) denotes the location belonging to the point whereto program control is transferred when receiving send $\in \mathcal{SEND}$ by rec $\in \mathcal{RECEIVE}$; the clause $\Box\neg$ *arrived*(send) expresses that this particular signal corresponding to send is uniquely identified (by the location, in our refined sense, of send) and therefore will never arrive anymore. The first axiom describes 1 above: when program control is at a receive case action, progress can only be caused by a time-out or the arrival of a signal. The second axiom then describes what happens when a time-out occurs and no signal has arrived and the third axiom describes what happens when a signal has arrived (i.e. it describes 2 above).

When P allows more than one signal to be receivable at a time, we can impose a second fairness condition, namely fairness relative to the receipt of signals (see §1.2).

Of course, a real programming language (e.g. complete CHILL) will probably offer more possibilities to receive signals. However, practical experience at PTI has shown that the incorporation of axioms for such specific cases is usually straightforward.

Other real-time features such as wait-constructs can be easily incorporated in an axiom system. E.g. for a wait action we can add the following axioms:

$\Box\forall \text{wait} \in \mathit{WAIT}$
$[\mathit{at}(\mathit{location}(\text{wait}))$
$\rightarrow [\mathit{at}(\mathit{location}(\text{wait}))\, \mathcal{U}_{=0}\, \mathit{at}(\mathit{timerstartlocation}(\text{wait}))$
$\wedge \Diamond_{=\mathit{waitvalue}(\text{wait})}\, \mathit{after}(\mathit{timerstartlocation}(\text{wait}))$
$\wedge \neg \Diamond_{<\mathit{waitvalue}(\text{wait})}\, \mathit{after}(\mathit{timerstartlocation}(\text{wait}))]]$

$\Box\forall \text{wait} \in \mathit{WAIT}$
$[\mathit{at}(\mathit{timerstartlocation}(\text{wait}))$
$\rightarrow \mathit{at}(\mathit{timerstartlocation}(\text{wait}))\, \tilde{\mathcal{U}}$
$\mathit{after}(\mathit{timerstartlocation}(\text{wait}))]$.

The notations in these axioms are analogous to those in the axioms for a receive case action. Comparing the first axiom above with the first axiom for the receive case action, we see that the only difference is that a timer expires after a certain period of time in the latter case while in the former the wait is ended after a certain period of time. The second axiom is needed to describe the situation where program control is really waiting at the wait action, i.e. program control stays at the same location while it is still waiting. Notice that the two axioms for the wait action could alternatively have been expressed in one:

$\Box\forall \text{wait} \in \mathit{WAIT}$
$[\mathit{at}(\mathit{location}(\text{wait}))$
$\rightarrow \mathit{at}(\mathit{location}(\text{wait}))\, \mathcal{U}_{=0}$
$[\mathit{at}(\mathit{timerstartlocation}(\text{wait}))\, \mathcal{U}_{=\mathit{waitvalue}(\text{wait})}$
$\mathit{after}(\mathit{timerstartlocation}(\text{wait}))]]$.

However, we consider the two axioms above easier to understand as they contain no nested until operators.

§5. A programming example

As an illustration of the two fairness conditions (fairness of the medium relative to the transmission of signals and fairness relative to the receipt of signals), we now give a programming example written in a very small CHILL-like programming language (see Appendix A for a complete syntax).

See next page for the programming example.

We first explain the syntax of this program.

A program consists of a single module with its body. A body consists of data statements followed by action statements. A data statement is either a declaration statement or a process definition statement. In this program the variables i1,i2,i3 and source are declared. Variables always range over process instances. p1,p2 and p3 are the names of the 3 process definitions of this program. A process definition consists of a body and it can have several active process instances at a time.

... stands everywhere in this program for an arbitrary sequence of terminating actions that do not interfere with the communication pattern already given in the example.

A signal can be sent by a send action. A send action must specify a signal identification (a number) and a process instance (the destination).

The do for ever action has an obvious meaning.

example:

```
MODULE

   DCL i1,i2,i3,source;

   p1: PROCESS;
      ···;
      SEND 1 TO i3;
      DO FOR EVER;
         RECEIVE CASE
            2: ···;
               SEND 1 TO i3;
         ELSE ···;
               SEND 1 TO i3;
         ESAC;
      OD;
   END;

   p2: PROCESS;
      ···;
      DO FOR EVER;
         RECEIVE CASE
            2: ···;
         ELSE AFTER 10
               ···;
               SEND 1 TO i3;
         ESAC;
      OD;
   END;

   p3: PROCESS;
      ···;
      DO FOR EVER;
         RECEIVE CASE SET source;
            1: ···;
               SEND 2 TO source;
         ESAC;
      OD;
   END;

   START p3 SET i3;
   START p1 SET i1;
   START p2 SET i2;

END;
```

A receive case action must specify a number (at least one) of signal identifications followed by a colon and some actions. We say that a signal is receivable for a receive case action if its signal identification is one of the signal identifications listed in the receive case action. If program control is at the beginning of a receive case action there are two possibilities: at least one receivable signal for the receive case action has arrived or no such signal has arrived. In the first case a unique signal (from the receivable signals for the receive case action) will be received and control transfers to the actions specified in the receive case action after the signal identification of the received signal. In the second case (no signal arrived) the action undertaken depends on the presence of an else part in the receive case action. If no else part is present control will stay at the receive case action until (possibly never!) a receivable signal for the receive case action arrives. If on the other hand an else part is present program control will be transferred to the actions in the else part. Real-time is incorporated in this minilanguage by allowing an additional after part in the else part which specifies for how long (measured in some real-time unit) the arrival of a receivable signal for the receive case action will be awaited before control is transferred to the actions in the else part. A further option for the receive case action is the identification of the sender of a signal. This is done by the *SET*-option (see p2). It is used here to send a signal back to the sender.

Processes can be started by a start action. A start action must specify the name of the process definition of which the started process becomes a process instance and a start action may optionally specify a (declared) variable to which the process instance of the started process is assigned (see the 3 start actions at the end of the example).

Now we can indicate the purpose of the two fairness conditions. In an axiom system based on the temporal logic of §2 (used in a development group of PTI) we can derive the following properties for this fragment:

1. in both i1 (the only instance of p1) and i2 (the only instance of p2) infinitely many signals 1 are sent (see below for an informal deduction hereof);

2. infinitely many signals 1, coming both from i1 and from i2, arrive at their destination, c.q. i3, the only instance of p3 (this follows from 1 and the fairness condition for the medium);

3. in the receive case action of i3 infinitely many signals 1 coming from i1 and infinitely many signals 1 coming from i2 are received (this follows from 2, the presence of the do for ever action in i3 and the fairness condition for the receipt of signals);

4. infinitely many signals 2 will be sent, both to i1 as well as to i2 (this follows from 3 and the fact that i3 sends its signals back to the sender):

We shall now informally derive point 1 above (the rest has already been informally deduced from point 1). The case is easy for i1 as in every execution of the do loop a signal is sent. This is however not the case for i2.

Now suppose that point 1 does not hold for i2. This means that only finitely many signals 1 are sent in i2. Because in the else part of the receive case action of i2 a signal 1 is sent, this else part is executed only a finite number of times. But since the receive case action of i2 will be executed infinitely often, this means that in i2 signals 2 must be received infinitely often. Therefore, signals 2 have been sent to i2 infinitely often.

The only place in the program where signals 2 are sent is i3 (the only instance of p3). However, i3 only sends a signal to some process after it has received a signal 1 from this process (see the structure of the receive case action in p3). This would mean that i2 has sent infinitely many signals 1. A contradiction!

Hence point 1 is also valid for i2.

§6. Future work

A proof system for our logic will be worked out and a soundness and completeness proof will be given. In particular a study of the consequences of the introduction of the operators $\blacklozenge$ and $\mathcal{U}_{=t}$ will be made. Furthermore, real-time arithmetic should be studied in our logic, e.g. $\Box_{=t}\Box_{=t'}P \leftrightarrow \Box_{=t+t'}P$ could be a theorem of our logical system.

Further study should be devoted to the application of the logical ideas presented here to several other areas such as practical languages concerning real-time processes and their (a)synchronous communication.

Acknowledgements

We would like to thank participants of a Dutch Working Group in Semantics, especially Peter van Emde Boas and Jan Bergstra from the Mathematical Centre in Amsterdam, and members of the Computer Science Department of the University of Nijmegen for their helpful comments and suggestions at the presentations of an earlier version of the paper. Furthermore, we are grateful to PTI Hilversum for providing financial support to one of the authors, R.Koymans, while working on this paper.

References

[ADA] "The Programming Language ADA, Reference Manual",
Lecture Notes in Computer Science 106,
Springer Verlag, New York, 1981.

[ASTESIANO & ZUCCA] Astesiano,E. and Zucca,E.
"Semantics of Distributed Processes Derived by Translation",
Proceedings of the 11th GI-Jahrestagung,
Informatik Fachberichte 50, pp. 78-87,
Springer Verlag, New York, 1981.

[BERNSTEIN & HARTER] Bernstein,A. and Harter Jr.,P.K.
"Proving Real-time Properties of Programs with Temporal Logic",
Proceedings of the 8th Symposium on Operating Systems Principles,
pp. 1-11, ACM Special Interest Group on Operating Systems,
Vol. 15 No. 5 December 1981, New York.

[BRINCH HANSEN] Brinch Hansen,P.
"Distributed Processes: A Concurrent Programming Concept",
CACM 21-11, pp. 934-941, 1978.

[CHILL] "CHILL Recommendation Z.200 (CHILL Language Definition)",
C.C.I.T.T. Study Group XI, The International Telegraph and Telephone
Consultative Committee, Geneva, November 1980.

[GERTH & DE ROEVER] Gerth,R.T. and de Roever,W.P.
"A Proof System for Concurrent ADA Programs",
to appear in Science of Programming.

[HOARE] Hoare,C.A.R.
"Communicating Sequential Processes",
CACM 21-8, pp. 666-677, August 1978.

[KUIPER & DE ROEVER] Kuiper,R. and de Roever,W.P.
"Fairness Assumptions for CSP in a Temporal Logic Framework",
TC2 Working Conference on the Formal Description of
Programming Concepts, Garmisch, June 1982.

[LAMPORT] Lamport,L.
" 'Sometime' is Sometimes 'NOT Never' ",
A Tutorial on the Temporal Logic of Programs,
SRI International CSL-86, January 1979.

[MANNA & PNUELI] Manna,Z. and Pnueli,A.
"How to Cook a Temporal Proof System for your Pet Language", Proceedings of the Symposium on Principles of Programming Languages, Austin, Texas, January 1983.

[MISRA,CHANDY & SMITH] Misra,J. ,Chandy,K.M. and Smith,T.
"Proving Safety and Liveness of Communicating Processes, with Examples", Proceedings of the ACM SIGACT-SIGOPS Conference on the Principles of Distributed Computing, Ottawa, Canada, August 1982.

[PNUELI & DE ROEVER] Pnueli,A. and de Roever,W.P.
"Rendezvous with ADA - A Proof Theoretical View", Proceedings of the AdaTEC Conference, Crystal City, 1982.

[SCHLICHTING & SCHNEIDER] Schlichting,R.D. and Schneider,F.B.
"Using Message Passing for Distributed Programming: Proof Rules and Disciplines", Proceedings of the ACM SIGACT-SIGOPS Conference on the Principles of Distributed Computing, Ottawa, Canada, August 1982.

Appendix A: the syntax of the minilanguage

To describe the context-free syntax of the minilanguage we use an extended Backus-Naur Form, in which [T] denotes that a syntactic element T is optional and { T }* denotes repetition of T zero or more times and { T }$^+$ denotes repetition of T one or more times.

<program> ::= [<name>:] *MODULE* <body> *END;*

<name> ::= <letter> { <letter> | <digit> }*

<letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<body> ::= <data statement list> <action statement list>

<data statement list> ::= { <data statement> }*

<data statement> ::= <declaration statement> |
                     <process definition statement>

<declaration statement> ::= *DCL* <name list>*;*

<name list> ::= <name> { ,<name> }*

<process definition statement> ::= <name>: *PROCESS;* <body> *END;*

```
<action statement list> ::= { <action statement> }*

<action statement> ::= <start action> | <send action> |
                       <receive case action> |
                       <do for ever action>

<start action> ::= START <name> [SET <location>];

<location> ::= <name>

<send action> ::= SEND <signal identification> TO <location>;

<signal identification> ::= <unsigned integer>

<unsigned integer> ::= { <digit> }+

<receive case action> ::=
        RECEIVE CASE [SET <location>;]
                { <signal receive alternative> }+
        [ELSE  [AFTER <timevalue>]
                <action statement list>]
        ESAC;

<signal receive alternative> ::=
        <signal identification>: <action statement list>

<timevalue> ::= <unsigned integer>

<do for ever action> ::= DO FOR EVER;
                             <action statement list>
                         OD;
```

Appendix B: interactions of the operator $\blacklozenge$

In this appendix we describe the interactions of the operator $\blacklozenge$ with all non real-time operators. Let P and Q be formulae, then the following holds:

1. $P \rightarrow \blacklozenge P$
   This is similar to $\Diamond$ ($\blacklozenge$ is the "dual" of $\Diamond$).

2. $\blacklozenge\blacklozenge P \leftrightarrow \blacklozenge P$
   This is similar to $\Diamond$.

3. $\blacklozenge\Box P \leftrightarrow \Box P$
   See 1 for the reverse implication. The forward implication holds as the future of the past includes the future of the present!

4. $\Box\blacklozenge P \leftrightarrow \blacklozenge P$
   The forward implication holds as $\Box Q \rightarrow Q$ is a valid formula. The reverse implication holds as the past in the present will also be the past in the future!

5. $\blacklozenge\Diamond P \leftrightarrow \Diamond\blacklozenge P \leftrightarrow \blacklozenge P \vee \Diamond P$
   This states that $\blacklozenge\Diamond P$ or $\Diamond\blacklozenge P$ is the same as P holding sometime (in past,at present or in future)!

6. $\blacklozenge(P \vee Q) \leftrightarrow \blacklozenge P \vee \blacklozenge Q$
   This is similar to $\Diamond$.

7. $\blacklozenge(P \wedge Q) \rightarrow \blacklozenge P \wedge \blacklozenge Q$
   This is similar to $\Diamond$ (the reverse implication is false).

8. $\neg\blacklozenge P \rightarrow \blacklozenge\neg P$
   This is similar to $\Diamond$ (the reverse implication is false). We can derive 8 from 1 by applying it twice (the first time in opposite direction): $\neg\blacklozenge P \rightarrow \neg P \rightarrow \blacklozenge\neg P$.

9. $\blacklozenge \exists v \in \mathcal{D}\ P(v) \leftrightarrow \exists v \in \mathcal{D}\ \blacklozenge P(v)$ for all sorts $\mathcal{D}$
This is similar to $\Diamond$.

10. $\blacklozenge \forall v \in \mathcal{D}\ P(v) \rightarrow \forall v \in \mathcal{D}\ \blacklozenge P(v)$ for all sorts $\mathcal{D}$
This is similar to $\Diamond$ (the reverse implication is false).

11. $(\blacklozenge P \rightarrow \blacklozenge Q) \rightarrow \blacklozenge (P \rightarrow Q)$
This is similar to $\Diamond$ (the reverse implication is false). We can derive 11 as follows: if $\neg \blacklozenge P$, then according to 8 $\blacklozenge \neg P$ and hence $\blacklozenge (P \rightarrow Q)$ (as $\neg P \rightarrow (P \rightarrow Q)$); suppose therefore that $\blacklozenge P$ holds, then according to the premiss we have $\blacklozenge Q$ and hence $\blacklozenge (P \rightarrow Q)$ (as $Q \rightarrow (P \rightarrow Q)$). We can furthermore remark that neither side of the equivalence is valid if we replace $\rightarrow$ by $\leftrightarrow$.

12. $\blacklozenge \bot \leftrightarrow \bot$

This is obvious as for every pair (σ, j) $(\sigma, j) \not\models \bot$.

13. $\blacklozenge \top \leftrightarrow \top$

This is obvious as $(\sigma, j) \models \neg \bot$, hence $(\sigma, j) \models \top$ for every pair (σ, j).

14. $\Diamond(P\,\mathcal{U}\,Q) \rightarrow \Diamond P\;\mathcal{U}\;\Diamond Q$

The reverse implication is false. We can derive 14 as follows: remark that $\Diamond(P\,\mathcal{U}\,Q) \leftrightarrow \Diamond Q \vee P\,\mathcal{U}\,Q$ and $\Diamond P\;\mathcal{U}\;\Diamond Q \leftrightarrow \Diamond Q \vee (\Diamond P)\,\mathcal{U}\,Q$ are valid; as $P\,\mathcal{U}\,Q \rightarrow (\Diamond P)\,\mathcal{U}\,Q$ 14 immediately follows.

15. $\Diamond(P\,\tilde{\mathcal{U}}\,Q) \rightarrow \Diamond P\,\tilde{\mathcal{U}}\,\Diamond Q$

We will derive this and show that the reverse implication is false. Notice that according to the definition of $\tilde{\mathcal{U}}$, respectively 6 and 3, the following holds:

$\Diamond(P\,\tilde{\mathcal{U}}\,Q) \leftrightarrow \Diamond(\Box P \vee P\,\mathcal{U}\,Q) \leftrightarrow$
$\leftrightarrow \Diamond\Box P \vee \Diamond(P\,\mathcal{U}\,Q) \leftrightarrow \Box P \vee \Diamond(P\,\mathcal{U}\,Q).$

For $\Diamond P\,\tilde{\mathcal{U}}\,\Diamond Q$, now according to the definition of $\tilde{\mathcal{U}}$, respectively 4, we have a similar property:

$\Diamond P\,\tilde{\mathcal{U}}\,\Diamond Q \leftrightarrow \Box\Diamond P \vee \Diamond P\,\mathcal{U}\,\Diamond Q \leftrightarrow \Diamond P \vee \Diamond P\,\mathcal{U}\,\Diamond Q.$

Because $\Box P \rightarrow P \rightarrow \Diamond P$ holds and $\Diamond(P\,\mathcal{U}\,Q) \rightarrow \Diamond P\;\mathcal{U}\;\Diamond Q$ holds according to 14, 15 immediately follows. Furthermore we see that the reverse implication of 15 is false.