10

A Unified Model for CSP-like Languages with Specifications

L. Lai
University of Bradford
Department of Computing, University of Bradford, Bradford, BD7 1DP, UK. Telephone: 01274-383913. Fax: 01274-383920.
email: l.m.lai@comp.bradford.ac.uk

Abstract

A unified model for CSP-like languages with specifications is presented. It is based on adding a specification statement to a CSP-like language. The extended language becomes thus a specification language and programs are viewed as a subclass of specifications. The specification statement is of the Hoare-triple form and can specify both safety and liveness properties. The semantics of the extended language is defined, which can serve as the foundations of refinement calculi and proof systems for CSP-like languages.

Keywords

CSP, Semantics, specification, communication, concurrency.

1 INTRODUCTION

A number of mathematical models already exist for CSP-like languages, such as Roscoe (1984), Zwiers (1989), and Pandya and Joseph (1991), etc.. A common salient feature among them is that they all support the compositional method: the specification of a concurrent system can be derived from the specifications of its subsystems independent of their implementation. However, few of them have been extended to include specifications which are essential for supporting a top-down refinement development method.

In Roscoe's model for occam (Roscoe, 1984), internal states have been added to the failures model for CSP (Brookes and Roscoe, 1984). A process is thus modelled as a set of triples (tr, ref, s), where s is a state. The initial states of a process is assumed undefined. A semantics for a large subset of occam was then defined. Roscoe's model, however, is inappropriate for modelling Hoare-triple specification formulae which specify the relationships between pre- and postconditions of state transitions. Variants of Roscoe's model are Hoare and Roscoe (1984), and He (1990), where a process is specified by a single predicate with four free variables: traces, refusals, states (including initial states), and status. Both models use a process status variable st to indicate termination, nontermination, and divergence. Finite alphabets are also used to eliminate unbounded nondeterminism. Consequently, the healthiness criteria in those two models are closely related to those of

G. V. Bochmann et al. (eds.), *Formal Description Techniques VIII*

ours. However, as those models are designed to support the one-predicate specification formulae, they are not suitable for defining the Hoare-triple specification statements.

Another model for CSP-like languages is Zwiers's (Zwiers, 1989). In this model, a process is modelled as a set of triples of the form, (s_0, tr, s). By employing only traces, only safety properties can be specified, where liveness properties of communicating processes are also important. However, this model has also been extended to include mixed terms, and our specification statement is inspired by Zwiers's (see Zwiers, 1989, 153-4).

In this paper a unified mathematical model for a CSP-like language with specifications is presented, which serves as the foundation for the study of specification and refinement development of such languages (Lai and Sanders, 1995). In contrast to CSP, our model aims to make internal states explicit. The main contribution of this paper is the extension of the existing models to mixed terms. Doing so, a uniform treatment is obtained for modelling both programs and specifications, or a mixture of both.

In Section 2 a model for communicating processes with states is presented. In Section 3 a CSP-like language CSPL is introduced and its semantics is defined. In Section 4 a specification statement is defined and CSPL is extended to include such specification statements; the semantics of the mixed terms are defined in the extended model. Section 6 gives an example of the specification statement. Section 7 concludes the paper.

2 THE DESCRIPTION OF PROCESSES

We assume that the reader is familiar with the basic concepts of CSP, such as traces tr and refusals ref. In this paper, refusals are sets of channel names instead of communication events. We assume the basic semantic domains as follows: the finite communication alphabet $(c \in) Chan$; the finite message set $(v, v' \in) Val$; the program variable set $(x, y \in) Var$; the communication-event set $Comm \mathrel{\widehat{=}} Chan \times Val$.

To model the communication behaviours of a process, CSP's failures model is one of the appropriate models (Brookes and Roscoe, 1984). To model state transitions, where the contents of program variables are of great concern, another observable is needed. A state $s : Var \to Val^+$ maps each variable in Var to a value in Val^+, where $Val^+ = Val \cup \{\bot_{Val}\}$. If a variable is mapped to $\bot_{Val}$, it is said to be undefined. The set of all such states is denoted $State$ and called proper states.

A process usually starts in an initial state s_0; after engaging in some communications and internal actions, it may terminate in some final state s. We therefore use a quadruple

$$(s_0, tr, ref, s)$$

to denote such a computation. A process is capable of doing many computations. Its behaviour can be described by a set of computations. For example, a successfully terminating process **skip**, with communication alphabet $Chan$, can be defined as

$$\mathbf{skip} \mathrel{\widehat{=}} \{(s_0, <>, ref, s_0) \mid s_0 \in State \wedge ref \subseteq Chan\}. \tag{1}$$

skip can start in any state s_0; it refuses to do anything but terminates with its state unchanged. The conjunct $ref \subseteq Chan$ in (1) is normally dropped as it is always true.

To model divergences, where a process engages in an infinite unbroken sequence of internal actions, a special state $\bot_{State}$, or simply $\bot$, is introduced. To model deadlocks, where

a process terminates unsuccessfully and never engages in any further communication, another special state $\top$ is used. Thus deadlock **stop** can be modelled as

$$\mathbf{stop} \triangleq \{(s_0, <>, ref, \top) \mid s_0 \in State\}. \tag{2}$$

The set of all computations is defined as

$$Comp \triangleq State \times Comm^* \times \mathcal{P}(Chan) \times State^+,$$

where A^* is the set of all the finite strings of A, $State^+ \triangleq State \cup \{\bot_{State}\} \cup \{\top\}$, and $\mathcal{P}(A)$ is the power set of A.

Definition 1 (Processes) *For a given finite communication alphabet Chan, a variable set Var, and a finite value set Val, the process space Proc is a set of all subsets P of Comp, which satisfies the following conditions: for any* $s_0 \in State$,

P1 $traces(P, s_0)(= \{tr \mid \exists s.\ (s_0, tr, \phi, s) \in P\})$ *is nonempty and prefix-closed:*

$$tr \neq <> \wedge (s_0, tr'^\wedge tr, ref, s) \in P \Longrightarrow (s_0, tr', \phi, \top) \in P;$$

P2 $(s_0, tr, ref, s) \in P \wedge ref' \subseteq ref \Longrightarrow (s_0, tr, ref', s) \in P;$
P3 $(s_0, tr, ref, s) \in P \wedge \neg \exists v, s'.\ (s_0, tr^\wedge <c.v>, \phi, s') \in P \Longrightarrow (s_0, tr, ref \cup \{c\}, s) \in P;$
P4 $(s_0, tr^\wedge <c_{in}.v>, ref, s) \in P \Longrightarrow \forall v' \exists s'.\ (s_0, tr^\wedge <c_{in}.v'>, ref, s') \in P;$
P5 $(s_0, tr, ref, s) \in P \wedge s \in State \Longrightarrow \forall ref'.\ (s_0, tr, ref', s) \in P;$
P6 $(s_0, tr, ref, \bot) \in P \Longrightarrow \forall tr', ref', s'.\ (s_0, tr^\wedge tr', ref', s') \in P,$

where v and v' range over Val, tr and tr' over Comm, ref and ref' over* $\mathcal{P}(Chan)$, *s and s' over* $State^+$, *and* c_{in} *is an input channel.* □

The informal meanings of the above conditions are as follows. **P1** states that the traces of a process from an initial state is nonempty and prefixed-closed; **P2** states that if a set of channels can be refused, so can any subset of it; **P3** states that a process will refuse a channel on which it cannot communicate any message after its current trace; **P4** states that selective input is not allowed, which means that, if an input channel cannot be refused, it must be prepared to input any value from its environment; **P5** states that if a process has a proper internal state after the current trace, it can terminate successfully in that state without doing any further communications; **P6** states that if a process diverges, it behaves chaotically afterwards.

The simplest process which satisfies the above conditions is also the worst:

$$\bot_{Proc} \triangleq Comp, \tag{3}$$

which may do anything and may refuse to do anything; it behaves chaotically.

We define an ordering relationship $\sqsubseteq$ among processes:

$$P_1 \sqsubseteq P_2, \quad \text{if and only if } P_2 \subseteq P_1.$$

Two processes are comparable in this ordering if the former can do every computation of the latter and possibly more, and the latter diverges less and more deterministic.

Theorem 1 $(Proc, \sqsubseteq, \bot_{Proc})$ *is a complete partial order.*

3 THE DENOTATIONAL SEMANTICS OF CSPL

In this section a CSP-like language (CSPL) is introduced and its denotational semantics is defined. As the objective of our study is to lay the mathematical foundations for the study of specification and refinement methods for CSP-like languages, CSPL is kept down to its essentials, i.e., assignment, communications and concurrency. The syntactic domain *Prog* of CSPL programs is defined as follows.

$$P \ ::= \ \textbf{skip} \mid \textbf{stop} \mid \textbf{div} \mid x := e \mid c?x \mid c!e \mid P_1 \ ; \ P_2 \mid P_1 \parallel P_2.$$

The intuitive meanings of these program constructs are as usual. $P_1 \parallel P_2$ consists of two processes P_1 and P_2 running in parallel, with all the communications on their linked channels being synchronized; it diverges when either P_1 or P_2 does and terminates only when both do.

The definitions for various expressions are omitted. The main semantic function is

$$[\![\bullet]\!] \ : \ \mathit{Prog} \rightarrow \mathit{Proc}.$$

The intuitive meaning of $[\![\bullet]\!]$ is that, given a program P, $[\![P]\!]$ produces a process on *Proc*. For expressions e, the value of e, evaluated in a state s, is denoted by $e(s_0)$. If anything goes wrong, such as a variable is undefined, the value of $e(s_0)$ is *error*. Whenever this happens, $[\![\bullet]\!]$ produces a divergent process.

The semantics of **skip**, **stop** and **div** are already given in (1), (2) and (3), respectively. The assignment statement $x := e$ is defined as follows.

$$[\![x := e]\!] \triangleq \{(s_0, <>, \mathit{ref}, s_0[e(s_0)/x]) \mid s_0 \in \mathit{State}\}, \qquad \text{where } e(s_0) \neq \mathit{error}$$

and $s[v/x]$ is the same as s except that the value of x is v.

The output process $c!e$ outputs one message e onto channel c and terminates with its state unchanged.

$$\begin{aligned}[\![c!e]\!] \ \triangleq \ & \{(s_0, <>, \mathit{ref}, \top) \mid s_0 \in \mathit{State} \wedge c \notin \mathit{ref}\} \\ & \cup \{(s_0, <c.v>, \mathit{ref}, s_0) \mid s_0 \in \mathit{State} \wedge v = e(s_0)\}, \qquad \text{where } e(s_0) \neq \mathit{error}.\end{aligned}$$

The semantics of the input process $c?x$ is similar to that of $c!e$ and left with interested readers.

If P_1 and P_2 are two processes with the same alphabet, $(P_1 \ ; \ P_2)$ is a process which behaves like P_1, except that if P_1 terminates successfully, it continues behaving like P_2.

$$\begin{aligned}[\![P_1 \ ; \ P_2]\!] \ \triangleq \ & \{(s_0, tr, \mathit{ref}, \top) \mid (s_0, tr, \mathit{ref}, \top) \in [\![P_1]\!]\} \\ & \cup \{(s_0, tr^\frown tr', \mathit{ref}', s) \mid (s_0, tr, \mathit{ref}, \bot) \in [\![P_1]\!]\} \\ & \cup \{(s_0, tr^\frown tr', \mathit{ref}, s) \mid \exists s'. \ (s_0, tr, \phi, s') \in [\![P_1]\!] \wedge (s', tr', \mathit{ref}, s) \in [\![P_2]\!]\}.\end{aligned}$$

The first two clauses say that any computation of P_1 that does not terminate is also a computation of $(P_1 \ ; \ P_2)$; the third clause says that, if P_1 terminates in a *proper* state in which P_2 can start, the execution of $(P_1 \ ; \ P_2)$ continues and it behaves like P_2.

The parallel composition $P_1 \parallel P_2$ is the key construct in a parallel language. We postulate that P_1 and P_2 do not share any program variable other than read-only ones and

a process cannot use a channel for both input and output. It is defined as

$$[\![P_1 \parallel P_2]\!] \;\widehat{=}\; \{(s_0, tr, ref_1 \cup ref_2, s_1 \oplus s_2) \mid \bigwedge_{i=1,2} (s_0, tr \restriction \alpha P_i, ref_i, s_i) \in [\![P_i]\!]\}$$
$$\bigcup_{i,j \in \{1,2\} \wedge i \neq j} \{(s_0, tr, ref, s) \mid \exists\, tr' \leq tr.\ (s_0, tr' \restriction \alpha P_i, \phi, \bot) \in [\![P_i]\!] \wedge$$
$$tr' \restriction \alpha P_j \in traces(P_j, s_0)\},$$

where $tr \restriction A$ is a trace obtained from tr by removing all the events that happened on channels not in A and $\oplus$ is defined as follows:

$$(s_1 \oplus s_2)(x) = \begin{cases} \top, & \text{if either } s_1 = \top \text{ or } s_2 = \top \\ s_1(x), & \text{otherwise, if } s_1 \neq \bot \\ s_2(x), & \text{otherwise, if } s_2 \neq \bot \\ \bot, & \text{otherwise} \end{cases}$$

If s_1 and s_2 map x into different non-$\bot$ values, the parallel combination is broken. The disjointness constraint of $\parallel$ guarantees that this won't happen.

Theorem 2 *All the processes defined above are well-defined and continuous.*

4 SPECIFICATION STATEMENTS

A specification of a communicating process with states describes not only its communication behaviours, but also the relationships between communications, initial states and final states. In this paper we take a novel view of communicating processes with states and regard them as generalized-state (tr, s) transformers (Zwiers, 1989): a process starts in an initial state (tr_0, s_0), does some communications in tr, and, if it terminates, it does so in a final state $(tr_0^\frown tr, s)$. Take the following process $P = (P_1 \ ; \ P_2)$ for example:

$$P :: \underbrace{c!1 \ ; \ x := 4}_{P_1} \ ; \ \underbrace{b!3 \ ; \ y := x}_{P_2}. \tag{4}$$

P takes an initial generalized-state $(<>, s_0)$ to a final one $(<c.1, b.3>, s_0[4/x, 4/y])$. It performs two communications, $c!1$ and $b!3$, and two state transitions, $x := 4$ and $y := x$. P_1 takes an initial state $(<>, s_0)$ to a final state $(< c.1 >, s_0[4/x])$. It does only one communication $c!1$ and one state transition $x := 4$. P_2, in the presence of P_1, takes an initial state $(< c.1 >, s_0[4/x])$ to a final state $(< c.1, b.3 >, s_0[4/y])$. It performs one communication $b!3$ and one state transition $y := x$. From P_2, we can see that, to describe a process with states, initial traces can be used so that it can be specified in the traditional predicate-transformer way, taking its environment into account (see Section 6).

The following predicates are used to describe the initial and final generalized-states, and the communication behaviour of a process.

$$pre \ : \ Comm^* \times State \rightarrow \{true, false, error\}$$
$$post \ : \ Comm^* \times Comm^* \times State \times State \rightarrow \{true, false, error\}$$

$$I \ : \ Comm^* \times Comm^* \times \mathcal{P}(Chan_{\surd}) \to \{true, false, error\}, \tag{5}$$

where "$\surd$" represents successful termination and $A_a = A \cup \{a\}$.

A special symbol "$\surd$" is added to the domain of the refusals of specification statements. If $\{\surd\}$ is a possible refusal of a specification statement after tr, it may refuse to terminate successfully. The precise meaning of "$\surd$" will be given when the semantics of the specification statement is defined.

The specification statement (so called in Morgan, 1994) is defined as follows.

Definition 2 (Specification statements) *A specification statement Sp, with the communication alphabet $\alpha(Sp)$ and a list of finite alterable variables w, is a quadruple*

$$Sp \ :: \ I, w : [pre, post], \tag{6}$$

where I, pre and $post$ are predicates of the types defined in (5). □

The intuitive meaning of a specification statement Sp is as follows. When started in one of the initial states satisfying $pre(tr_0, s_0)$, Sp must be able to engage in any communications tr satisfying $\exists\, ref.\ I(tr_0, tr_0^\frown tr, ref)$; if Sp terminates, it does so in one of the final states satisfying $post(tr_0, tr_0^\frown tr, s_0, s)$, with the terminating trace tr satisfying $I(tr_0, tr_0^\frown tr, \phi)$; if Sp cannot start in an initial state s_0, i.e., $\forall\, tr_0.\ \neg pre(tr_0, s_0)$, it diverges immediately.

5 MIXED TERMS

The development of a program usually starts with a specification or abstract program and ends with an executable program or code. In general, during the development, we have "programs" in which both specifications and code appear. We regard these "programs" as mixed terms (Morgan, 1994).

We expand the program domain $Prog$ in Section 3 to mixed terms $MProg$.

$$M ::= I, w : [pre, post] \mid \mathbf{skip} \mid \mathbf{stop} \mid \mathbf{div} \mid x := e \mid c?x \mid c!e \mid M_1 \ ; \ M_2 \mid M_1 \parallel M_2.$$

The semantics of mixed terms can also be defined as a set of computations. Assuming the same semantic domains and refinement ordering as those in Section 2, we have

Theorem 3 *$(\mathcal{P}(Comp), \sqsubseteq, \perp_{MixT})$ is a complete partial order.*

We use the same symbol $[\![\bullet]\!]$ as that in Section 3 for the main semantic function:

$$[\![\bullet]\!] : MProg \to \mathcal{P}(Comp).$$

The semantics for mixed terms are the same as those defined in Section 3, except that for the specification statement Sp.

$$\begin{aligned} & [\![I, w : [pre, post]]\!] \\ \widehat{=} \quad & \{(s_0, tr, ref, s) \mid \forall\, tr_0.\ pre(tr_0, s_0) \Longrightarrow I(tr_0, tr_0^\frown tr, \phi) \wedge s_0 =_w s \wedge \\ & \qquad\qquad post(tr_0, tr_0^\frown tr, s_0, s) \wedge ref \subseteq Chan\} \end{aligned} \tag{7}$$

$$\cup \{(s_0, tr, ref, \top) \mid \forall tr_0.\ pre(tr_0, s_0) \Longrightarrow I(tr_0, tr_0^\frown tr, ref \uplus \{\surd\})\} \quad (8)$$
provided that *pre*, *post*, and I do not evaluate to *error*

where $(s_1 =_w s_2) \mathrel{\widehat{=}} \forall x \notin w.\ s_1(x) = s_2(x)$ and $A \uplus \{a\} \mathrel{\widehat{=}} A \cup \{a\}$, where $a \notin A$.

(7) defines the terminating computations of Sp, as well as its divergent computations. The specification statement is a total-correctness formula in the sense that, if Sp has some terminating computation (s_0, tr, ref, s), there must exist some initial trace tr_0 such that $pre(tr_0, s_0)$ holds for s_0 and its terminating trace tr and final state s, together with the initial trace tr_0, satisfies the communication invariant $I(tr_0, tr_0^\frown tr, \phi)$ and the postcondition $post(tr_0, tr_0^\frown tr, s_0, s)$. The state transition from s_0 to s can only be achieved by changing those alterable variables in w. Termination is interpreted as a nondeterministic choice. Therefore, for Sp to terminate successfully if there exists some proper final state, we have $ref \subseteq Chan$ in (7). If there is no initial trace satisfying $pre(tr_0, s_0)$ for an initial state s_0, then Sp diverges; in this case, (7) $= \bot_{MixT}$.

(8) defines the nonterminating computations of Sp. To define nonterminating computations precisely, a new element "$\surd$" is introduced into the domain of refusals in the communication invariant I. Without it, unwanted nonterminating computation $(s_0, tr, ref, \top)$ would be included in $[\![Sp]\!]$ for every terminating computation (s_0, tr, ref, s) in $[\![Sp]\!]$. If Sp diverges in some s_0, then (8) $\subseteq$ (7).

Theorem 4 *All the constructors defined above (including those defined in Section 3) are continuous on* $\mathcal{P}(Comp)$.

6 AN EXAMPLE

We specify the three processes in (4) in the following, which have the same communication alphabet $\{b, c\}$ and program variables x and y.

P changes both x and y, communicates $c!1$ and $b!3$, and terminates. It is specified by

$$I, \{x, y\} : [tr =< >,\ \ tr =<c.1, b.3> \wedge x = y = 4], \quad (9)$$

where $I \mathrel{\widehat{=}} (tr =< > \wedge c \notin ref) \vee (tr =<c.1> \wedge b \notin ref) \vee (tr =<c.1, b.3> \wedge \surd \notin ref)$.

P_1 is similar to P, but changes only x and communicates $c!1$. It is specified by

$$I_1, \{x\} : [tr =< >,\ \ tr =<c.1> \wedge x = 4], \quad (10)$$

where $I_1 \mathrel{\widehat{=}} (tr =< > \wedge c \notin ref) \vee (tr =<c.1> \wedge \surd \notin ref)$.

P_2 changes only y and communicates $b!3$. By choosing an initial trace $tr_0 =< c.1 >$ for the precondition of P_2, we can retain both the communication invariant I and the postcondition of (9) for P_2, which will be very convenient for the sequential decomposition of P into P_1 and P_2:

$$I, \{x, y\} : [tr =<c.1> \wedge x = 4,\ \ tr =<c.1, b.3> \wedge x = y = 4]. \quad (11)$$

7 CONCLUSION

This paper has presented a unified model for CSP-like languages with specifications. It is based on the failures model for CSP in Brookes and Roscoe (1984), but designed with specifications in mind. The state transitions in our model is modelled as a total relation. The partial ordering $\sqsubseteq$ on *Proc* is also consistent with that in the failures model. Our model is extended to include mixed terms. The process constructors are proved to be well-defined and continuous on the extended model. A denotational semantics is given to the mixed terms.

Our specification statement is inspired by Morgan's for sequential programs (Morgan, 1994). It can specify certain liveness properties of a communicating process, such as nondeterminism and deadlocks. However, as CSP is not suitable for specifying fairness properties, nor is our specification statement. A common approach to do that is temporal logic (Pandya and Joseph, 1991). The investigation into a temporal logic method is a future research topic. Developing a refinement calculus based on this unified model for CSP-like languages is also of great interest (Lai and Sanders, 1995).

Acknowledgment
The author thanks Jeff Sanders, Jifeng He, and Carroll Morgan for their comments.

REFERENCES

Brookes, S.D. and Roscoe, A.W. (1984) An improved failures model for communicating processes. *Lecture Notes in Computer Science*, **197**, 281-305.

He, J. (1990) Specification-oriented semantics for ProCos level 0 language. Esprit BRA 3104, ProCos, Computing Laboratory, Oxford University.

Hoare, C.A.R. and Roscoe, A. W. (1984) Programs as executable predicates. In *Proceedings of the Fifth Generation Computer Systems 1984* (ed. ICOT), 220-8.

Lai, L. and Sanders, J. (1994) A refinement calculus for communicating processes with state. Submitted for publication.

Morgan, C. (1994) *Programming from Specifications.* Prentice Hall International, London, 2nd edition.

Pandya, P.K. and Joseph, M. (1991) P-A logic - a compositional proof system for distributed programs. *Distributed Computing (1991)***5**, 37-54.

Roscoe, A. W. (1984) Denotational semantics for occam. *Lecture Notes in Computer Science*, **197**, 306-29.

Zwiers, J. (1989) Compositionality, concurrency and partial correctness. *Lecture Notes in Computer Science*, **321**.

LUMING LAI received his D.Phil degree from the University of Oxford in 1993 and is since a lecturer in the Department of Computing at the University of Bradford, UK. His current research interests include specification, refinement, concurrency, and semantics.