Information Processing Letters 26 (1987/88) 165–171
North-Holland
4 December 1987

TWO NORMAL FORM THEOREMS FOR CSP PROGRAMS *

K.R. APT

Centre for Mathematics and Computer Science, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Luc BOUGÉ **

Département de Mathématiques et d'Informatique, Laboratoire d'Informatique de l'Ecole Normal Supérieure (LIENS), 45, rue d'Ulm, F-75230 Paris Cédex 05, France

Ph. CLERMONT

ETCA, Service CTME/OP, 16 bis av. Prieur de la Côte d'Or, F-94114 Arcueil, France

Communicated by David Gries
Received 20 February 1985
Revised 29 July 1985 and 13 July 1987

We define two normal forms for CSP programs. In the First Normal Form, each process contains only one I/O repetitive command and all its I/O commands appear as guards of this command. In the Second Normal Form, all guards of this I/O repetitive command are I/O guards. We describe an inductive method that transforms any CSP program into an equivalent program in first or second normal form. The notion of equivalence is discussed. It is shown that no transformation into second normal form can preserve deadlock freedom.

Keywords: CSP, normal form, partially interpreted computation, syntactic equivalence, deadlock freedom

1. Introduction

One of the best known theorems in Theoretical Computer Science states that every while program is equivalent to a program with one loop (see, e.g., [8]). We prove a similar result for CSP programs [9]. We exhibit two normal forms to which every CSP program can be brought. A CSP program is in a normal form (a *normal* program, in short) if each of its component processes contains only one I/O repetitive command and all its I/O commands appear as guards in this command. There are various reasons why the study of our normal programs can be of interest.

(1) *Program construction*—In the case of CSP programs, as opposed to while programs, several algorithms can be naturally expressed as normal programs. For example, most solutions to the distributed termination problem of Francez [5] are normal programs (see, e.g., [6,3]). For other algorithms written as normal programs, see for example [4].

(2) *Verification*—We found (see [1]) that a very simple proof system allows us to prove correctness of normal programs. Moreover, Queille and Sifakis [12] built a system allowing an automatic verification of finite-state normal programs. Adding to their system a preprocessor implementing the transformations described in this paper allows us to extend the use of their system to arbitrary finite-state CSP programs.

(3) *Event-driven computing*—In an event-driven concurrent system, local actions are triggered by

* This work was partially supported by CNRS Project C^3.
** L. Bougé is also affiliated with the Laboratoire d'Informatique, Université d'Orléans, B.P. 6759, F-45067 Orléans Cédex 02, France.

the occurrence of external events. This type of computing is typical in the case of network protocols. It is often modeled by means of interacting automata (see, e.g., [7]). Normal programs exhibit a structure that makes this view of distributed computing more explicit as each process alternates between communications and resulting local actions.

Equivalence of concurrent programs is a delicate and difficult issue. In the last section of this paper we analyze the notion used here and indicate its limitations.

2. Normal forms in CSP

We assume the reader to be familiar with CSP [9]. We consider a variant of CSP without nested parallelism and where output guards are allowed. We do not consider the Distributed Termination Convention. For simplicity, we omit all declarations.

A CSP program P is a parallel composition of named processes that operate on disjoint memories:

$$[P_1 :: S_1 \parallel \ldots \parallel P_k :: S_k].$$

Each process S is generated by the following grammar ($m \geqslant 1$):

$$S ::= \mathsf{skip} \mid cmd \mid \alpha \mid S_1 ; S_2 \mid \left[\mathop{\square}_{j=1}^{m} G_j \to S_j\right] \mid *\left[\mathop{\square}_{j=1}^{m} G_j \to S_j\right].$$

$$G ::= b \mid b ; \alpha.$$

Here, *cmd* denotes an assignment, b a Boolean expression, and α an I/O command. If a guard G is of the form b, then it is called a *purely Boolean guard*, otherwise an *I/O guard*. $Bool(G)$ denotes the Boolean part of a guard. A guard G is *enabled* when control is in front of it and $Bool(G)$ evaluates to true. In the sequel, $b_1 ; b_2$ denotes the conjunction $b_1 \wedge b_2$ of Boolean expressions.

2.1. Definition. A process S is in *first normal form* if it is of the form S_0 or

$$S_0 ; *\left[\mathop{\square}_{j=1}^{m} G_j \to S_j\right],$$

where $m \geqslant 1$ and none of the S_j contains an I/O command.

2.2. Definition. A process S is in *second normal form* if it is of the form S_0 or

$$S_0 ; *\left[\mathop{\square}_{j=1}^{m} G_j \to S_j\right],$$

where $m \geqslant 1$, none of the S_j contains any I/O command and, moreover, *all* of the G_j are I/O guards.

A CSP program is in first (respectively second) normal form if all its component processes are.

3. The transformations

3.1. First normal form

We now describe a function NF_1 that transforms each process S into a process $S' = NF_1(S)$ in first normal form. We proceed by induction on the structure of S. We assume an infinite set Z of fresh Boolean variables $z_1, z_2, \ldots$. We omit indices when no confusion can arise.

Base case. When S is skip or an atomic assignment command, S is already in first normal form, so $NF_1(S) = S$.

I/O command. Suppose S is an I/O command α. Select a fresh variable z from Z and define

$$NF_1(S) = z := \mathsf{true} ; *[z ; \alpha \to z := \mathsf{false}].$$

Sequential composition. Suppose S is of the form $S_1 ; S_2$. By induction, we have

$$NF_1(S_i) = Init_i ; *\left[\mathop{\square}_{j=1}^{m_i} G_j^i \to S_j^i\right]$$

where we can assume that sets Z_1 and Z_2 of fresh variables for each S_i are disjoint. Let z_1 and z_2 be two variables of $Z \setminus Z_1 \cup Z_2$. Define $NF_1(S)$ as follows:

$$Init_1 ; z_1 := \mathsf{true};\ z_2 := \mathsf{false};$$

$$*\left[\mathop{\square}_{j=1}^{m_1} z_1 ; G_j^1 \to S_j^1 ; TEST \quad \mathop{\square}_{j=1}^{m_2} z_2 ; G_j^2 \to S_j^2\right]$$

where $TEST$ stands for

$$\left[\begin{array}{l}\bigsqcup_{j=1}^{m_1} Bool(G_j^1) \to \mathsf{skip} \\ \square \bigwedge_{j=1}^{m_1} \neg Bool(G_j^1) \to z := \mathsf{false}; \\ z_2 := \mathsf{true};\ Init_2\end{array}\right].$$

Intuitively, z_1 is true when control is still in S_1 and z_2 is true when control is in S_2.

Repetitive command. Suppose S is of the form

$$*\left[\bigsqcup_{i=1}^{m} H_i \to R_i\right].$$

By induction, we have

$$NF_1(R_i) = Init_i;\ *\left[\bigsqcup_{j=1}^{m_i} G_j^i \to S_j^i\right]$$

where we can assume without loss of generality that sets Z_i of fresh variables for each R_i are pairwise disjoint. Let $turn_i$, $i = 1, 2, \ldots, m$ be fresh variables of $Z \setminus \bigcup_{i=1}^{m} Z_i$. Define $NF_1(S)$ as follows:

$$turn_1 := \mathsf{false}; \ldots; turn_m := \mathsf{false};$$

$$*\left[\begin{array}{l}\bigsqcup_{i=1}^{m} \bigwedge_{i=1}^{m} \neg turn_i;\ H_i \to turn_i := \mathsf{true}; \\ \qquad Init_i;\ TEST_i \\ \bigsqcup_{\substack{i=1,2,\ldots,m \\ j=1,2,\ldots,m_i}} turn_i;\ G_j^i \to S_j^i;\ TEST_i\end{array}\right]$$

where $TEST_i$ stands for

$$\left[\begin{array}{l}\bigsqcup_{k=1}^{m_i} Bool(G_k^i) \to \mathsf{skip} \\ \square \bigwedge_{k=1}^{m_i} \neg Bool(G_k^i) \to turn_i := \mathsf{false}\end{array}\right].$$

Intuitively, $turn_i$ holds when control is in subprogram R_i. Then, $TEST_i$ indicates whether R_i is terminated and $turn_i$ is reset to false if this is the case.

Alternative command. Suppose S is of the form

$$\left[\bigsqcup_{i=1}^{m} H_i \to R_i\right].$$

By induction, we have

$$NF_1(R_i) = Init_i;\ *\left[\bigsqcup_{j=1}^{m_i} G_j^i \to S_j^i\right]$$

in first normal form. Using a new variable z from $Z \setminus \bigcup_{i=1}^{m} Z_i$, we first transform S into the following process S':

$$z := \mathsf{true};$$

$$\left[\bigsqcup_{i=1}^{m} Bool(H_i) \to \mathsf{skip}\right];$$

$$*\left[\bigsqcup_{i=1}^{m} z;\ H_i \to R_i;\ z := \mathsf{false}\right].$$

Those two processes are related as follows. First suppose that S fails. This occurs when all conditions $Bool(H_i)$ evaluate to false initially. Then, S' fails much in the same way. Suppose S does not fail; then at least one of those conditions evaluates to true. In S', the alternative command then boils down to skip. In the repetitive command, the conditions are evaluated again and yield the same results as before, because processes operate on disjoint memories. At least one of them is thus guaranteed to evaluate to true, S' does not fail either, and behaves subsequently like S. $NF_1(S)$ is the result of applying transformation NF_1 to process S'.

This concludes the presentation of the transformation NF_1.

Property A. *For each process S, $NF_1(S)$ is a process in first normal form. The only atomic commands in $NF_1(S)$ in which variables from Z appear are of the form $z :=$ true or $z :=$ false.*

3.2. Second normal form

We now describe a procedure NF_2 that transforms each process S in first normal form into a process $NF_2(S)$ in second normal form. A process S in first normal form all of whose external guards contain an I/O command (or that contains no I/O command) is already in second normal form and we put $NF_2(S) = S$. Otherwise, it can be written as

$$Init;$$
$$*\left[\mathop{\square}_{i=1}^{m} G_i \to S_i \quad \mathop{\square}_{j=1}^{n} H_j \to T_j\right]$$

with $m > 0$ and $n > 0$, where all guards G_i are purely Boolean and all guards H_j do contain an I/O command. Let $CHOOSE$ be the following command:

$$turn_1 := \mathsf{false}; \ldots ; turn_n := \mathsf{false};$$
$$*\left[\mathop{\square}_{i=1}^{m} \bigwedge_{k=1}^{n} \neg turn_k ; G_i \to S_i \quad \mathop{\square}_{j=1}^{n} \bigwedge_{k=1}^{n} \neg turn_k ; Bool(H_j) \to turn_j := \mathsf{true}\right].$$

Execution of $CHOOSE$ consists of some iterations of the repetitive command

$$*\left[\mathop{\square}_{i=1}^{m} G_i \to S_i\right]$$

which contains no I/O command followed by the selection of an I/O guard H_j, provided its Boolean part $Bool(H_j)$ evaluates to true. We then define $NF_2(S)$ to be the following process:

$$Init; CHOOSE;$$
$$*\left[\mathop{\square}_{j=1}^{n} turn_j ; H_j \to T_j ; CHOOSE\right].$$

Observe that $Bool(H_j)$ is evaluated twice, once within $CHOOSE$ and then again within H_j. Both evaluations return necessarily the same result because processes operate on disjoint memories.

Property B. *For each process S in first normal form, $NF_2(S)$ is in second normal form. The only atomic commands in $NF_2(S)$ in which variables from Z appear are of the form $z :=$ true and $z :=$ false.*

3.3. Homogeneous processes

For certain processes, it is possible to describe a direct transformation that yields a process in second normal form. A process is *homogeneous* if, in each repetitive or alternative command, either all guards are purely Boolean or all guards contain an I/O command. Observe that a homogeneous process is in first normal form if and only if it is in second normal form. If we can modify procedure NF_1 so as to preserve homogeneity, then it will transform homogeneous processes into processes in *second* normal form. The only part of NF_1 that does not preserve homogeneity is the one dealing with a repetitive command S

$$*\left[\mathop{\square}_{i=1}^{m} H_i \to R_i\right]$$

whose guards are all purely Boolean. In this case, let $SWITCH$; be

$$\left[\mathop{\square}_{i=1}^{m} H_i \to turn_i := \mathsf{true}; Init_i \quad \square \bigwedge_{i=1}^{m} \neg H_i \to \mathsf{skip}\right].$$

Then, assuming the notation used in $NF_1(S)$, the transformed process is

$$turn_1 := \mathsf{false}; \ldots ; turn_m := \mathsf{false};$$
$$SWITCH;$$
$$*\left[\mathop{\square}_{\substack{i=1,2,\ldots,m \\ j=1,2,\ldots,m_i}} turn_i ; G_j^i \to S_j^i ; TEST_i ; \quad [turn_i \to \mathsf{skip} \square \neg turn_i \to SWITCH]\right].$$

Denote this modified transformation by NF_1'. Here, variables $turn_i$ are used for the same purpose as before. Setting a variable $turn_i$ to true can take place in the *SWITCH* command only.

Property C. *For each homogeneous process S,* $NF_1'(S)$ *is in second normal form. The only atomic commands in* NF_1' *in which variables from Z appear are of the form* $z :=$ true *and* $z :=$ false.

4. A notion of equivalence

We now wish to make precise in what sense every process S is equivalent to the process S' generated in Section 3 by the transformations NF_1, NF_2, and NF_1'. To this purpose, we first associate with each process S a regular language $L(S)$. Intuitively, $L(S)$ is the set of all uninterpreted possible computations of process S according to Plotkin's semantics [11].

The language $L(S)$ is over the alphabet consisting of atomic actions *cmd*, I/O commands α, Boolean conditions b, plus two special tokens $\langle skip \rangle$ and $\langle fail \rangle$ that denote respectively termination and failure. $L(S)$ is defined inductively as follows.

$$L(\mathsf{skip}) = \{\langle skip \rangle\},$$

$$L(cmd) = \{\langle cmd \rangle\},$$

$$L(\alpha) = \{\langle \alpha \rangle\},$$

$$L(G) = \begin{cases} \{\langle b \rangle \langle \alpha \rangle\} & \text{if } G = b;\ \alpha, \\ \{\langle b \rangle \langle skip \rangle\} & \text{if } G = b, \end{cases}$$

$$L(S_1; S_2) = L(S_1).L(S_2),$$

$$L([G_1 \to S_1 \,\Box \cdots \Box\, G_m \to S_m])$$
$$= [(L(G_1).L(S_1)) \cup \cdots \cup (L(G_m).L(S_m))] .\{\langle Bool \rangle \langle fail \rangle\},$$

$$L(*[G_1 \to S_1 \,\Box \cdots \Box\, G_m \to S_m])$$
$$= [(L(G_1).L(S_1)) \cup \cdots \cup (L(G_m).L(S_m))]^* .\{\langle Bool \rangle \langle skip \rangle\},$$

where Bool stands for

$$\neg Bool(G_1) \wedge \cdots \wedge \neg Bool(G_m).$$

Observe how the appropriate exit conditions are reflected.

To obtain the desired equivalence, we partially interpret the computations by evaluating the commands and conditions associated with auxiliary variables. In the processes generated by the transformations of Section 3, they can be of the following type exclusively:

$$z := \mathsf{true},$$

$$z := \mathsf{false},$$

$$B(z_1, \ldots, z_m, b_1, \ldots, b_n),$$

where B is some Boolean combination of its arguments. When evaluating the condition, variables z_i are substituted with their current value, *true* or *false*. The condition is said to be *unsatisfiable* if the resulting formula is equivalent to *false* as a formula of the predicate calculus with variables b_j. The condition is *satisfiable* if it is not unsatisfiable. Then we exclude contradictory computations, i.e., those that violate the rule that the selected Boolean conditions are all satisfiable. Finally, we erase all *skip's* and assignments $z :=$ true and $z :=$ false to auxiliary variables. Also, we merge adjacent Boolean formulas into their Boolean conjunction and reduce the resulting formula to some normal form (say, a conjunction of disjunctions for definiteness). Tautologies are then erased. Let $L'(S)$ be the resulting language. It is a language over the alphabet consisting of atomic actions *cmd*, I/O commands α, Boolean formula $B(b_1, \ldots, b_n)$, and token $\langle fail \rangle$. We say that two processes S_1 and S_2 are *equivalent* with respect to a set Z of auxiliary variables if

$$L'(S_1) = L'(S_2).$$

This equivalence can be best understood with an example. Consider the processes

$$S_1 = \alpha$$

and

$$S_2 = z := \mathsf{true};\ *[z;\ \alpha \to z := \mathsf{false}].$$

Then

$$L(S_1) = L'(S_1) = \{\langle \alpha \rangle\}.$$

On the other hand,

$$L(S_2) = \langle z := \text{true}\rangle(\langle z\rangle\langle\alpha\rangle\langle z := \text{false}\rangle)^* \langle\neg z\rangle\langle skip\rangle.$$

After the interpretation of actions related to the variable z we obtain the following set of words:

$$\{\langle z := \text{true}\rangle\langle true\rangle\langle\alpha\rangle\langle z := \text{false}\rangle\langle\neg false\rangle\langle skip\rangle, \langle z := \text{true}\rangle\langle true\rangle\langle\alpha\rangle\langle z := \text{false}\rangle (\langle false\rangle\langle\alpha\rangle\langle z := \text{false}\rangle)^* \langle\neg false\rangle\langle skip\rangle\}.$$

Here, only the computation

$$\langle z := \text{true}\rangle\langle true\rangle\langle\alpha\rangle\langle z := \text{false}\rangle\langle\neg false\rangle\langle skip\rangle$$

is not contradictory. Deleting from it all assignments to the variables of Z and *skip*'s, reducing sequences of adjacent Boolean formulas to their normal form and erasing tautologies, we get $\langle\alpha\rangle$ as desired.

We have the following theorems, whose tedious but straightforward proofs are omitted.

4.1. Theorem. *Both $NF_1(S)$ and $NF_1'(S)$ are equivalent to S with respect to set Z of auxiliary variables.*

4.2. Theorem. *$NF_2(S)$ is equivalent to S with respect to set Z of auxiliary variables.*

These equivalences are on the level of processes considered in isolation. The following theorem states some of its semantic consequences. By a *state* we mean a function assigning values to each of the variables. We consider $\perp$ as a special state indicating divergence. Given a CSP program P, we define its meaning $\mathcal{M}[11]$ by

$$\mathcal{M}[11](\sigma) = \{\tau, \tau \text{ is the final state of a properly terminating computation starting in state } \sigma\} \cup \{\perp, \text{there exists a diverging computation of } P \text{ starting in state } \sigma\}.$$

For two sets Σ_1 and Σ_2, and a set of variables Z we define

$$\Sigma_1 = \Sigma_2 \bmod Z \quad \text{iff } \{\sigma\backslash Z, \sigma\in\Sigma_1\} = \{\sigma\backslash Z, \sigma\in\Sigma_2\}$$

where $\sigma\backslash Z$ is the restriction of σ to the variables not in Z. We now say that two programs CSP P_1 and P_2 are *equivalent modulo Z* if, for all states σ,

$$\mathcal{M}[P_1](\sigma) = \mathcal{M}[P_2](\sigma) \bmod Z.$$

Note that this equivalence definition does not take into account possible deadlocks. We can finally state the appropriate theorem.

4.3. Theorem. *Let S_1 and S_2 be two equivalent processes with respect to a set Z of auxiliary variables. Let*

$$\mathscr{C} = [Q_1 :: T_1 \parallel \cdots \parallel Q_k :: [\ ] \parallel \cdots \parallel Q_n :: T_n]$$

be a context, and let $P_i = \mathscr{C}[S_i]$, $i = 1, 2$, be the CSP programs obtained by plugging process S_i into the context $\mathscr{C}$. Then, P_1 and P_2 are equivalent modulo Z.

Thus, up to deadlock, P_1 and P_2 exhibit the same functional behaviors.

5. Discussion

The equivalence relation introduced in Section 4 seems at first sight very strong, since it is basically a syntactic equivalence. However, it is concerned only with some form of traces (in the sense of [10]) of computations. Semantically, it assures only Theorem 4.3. In particular, it does not capture all relevant semantic properties naturally associated with concurrent programs, like deadlock freedom.

Indeed, consider two processes S and S' where

$$S = [\text{true}; Q?x \to \text{skip} \ \Box\text{true}; Q!x \to \text{skip}],$$
$$S' = [\text{true} \to Q?x; \text{skip} \ \Box\text{true} \to Q!x; \text{skip}].$$

Then, S and S' are equivalent in the sense of Section 4. However, the program

$$[P :: S \parallel Q :: P?y]$$

cannot deadlock whereas the program

$$[P :: S' \parallel Q :: P?y]$$

can. Thus, plugging equivalent processes in the same context, here

$$[P :: [\ ] \| Q :: P?y],$$

can yield two programs that behave differently. We can prove that Theorem 4.2 cannot be strengthened so that deadlock freedom is preserved in the above sense. This follows from the following theorem.

5.1. Theorem. *Let* $[P :: S \| Q :: T]$ *be a program in second normal form. Suppose that it admits two properly terminating computations,* C_1 *with some communication and* C_2 *without any communication. Then, it admits a deadlocked computation.*

Proof. Construct the deadlocked computation as follows. First take all steps carried out by P in C_1 until the I/O command selected for its first communication is reached. Then, append to it all steps carried out by Q in C_2. In the resulting computation, Q properly terminates whereas P reaches an entry to a repetitive command with all guards containing an I/O guard. Thus, a deadlock arises (observe that this would not necessarily hold if the Distributed Termination Convention of CSP were used). □

This shows that first normal form cannot be reduced to second normal form when deadlock freedom is to be preserved. This can be interpreted as a statement that use of nonhomogeneous guards strictly increases the expressive power of CSP.

Note

The first version of this paper appeared as a report [2]. After having written the present version, we learned of a related work by Zöbel [13]. Zöbel proposes transformations similar to ours, but does not elaborate on the underlying notion of equivalence.

References

[1] K.R. Apt, Correctness proofs of distributed termination algorithms, ACM Trans. Programm. Languages & Systems 8 (3) (1986) 388–405.

[2] K.R. Apt and Ph. Clermont, Two Normal Form Theorems for CSP Programs, Rept. No. RC 10975, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1985.

[3] K.R. Apt and J.-L. Richier, Real time clocks versus virtual clocks, in: Proc. Internat. Summer School on Control Flow and Data Flow: Concepts of Distributed Programming, NATO ASI Series F14 (Springer, Berlin, 1985).

[4] L. Bougé, Genericity and Symmetry for Distributed Systems: The Case of CSP, Thèse d'État, Univ. Paris 7, 1987; Rept. No. 87/2, LIENS, Paris, 1987 (in French).

[5] N. Francez, Distributed termination, ACM Trans. Programm. Languages & Systems 2 (1) (1980) 42–55.

[6] N. Francez, M. Rodeh and M. Sintzoff, Distributed termination with interval assertion, Proc. Internat. Coll. on Formalization of Programming Concepts, Peniscola, Spain, Lecture Notes in Computer Science, Vol. 107 (Springer, Berlin, 1981).

[7] M.G. Gouda, Closed covers: To verify progress for communicating finite state machines, IEEE Trans. Software Engrg. SE-10 (6) (1984) 846–855.

[8] D. Harel, On folk theorems, Comm. ACM 23 (7) (1980) 379–389.

[9] C.A.R. Hoare, Communicating sequential processes, Comm. ACM 21 (8) (1978) 666–677.

[10] C.A.R. Hoare, Some properties of predicate transformers, J. ACM 25 (3) (1978) 461–480.

[11] G. Plotkin, An operational semantics for CSP, in: D. Bjørner, ed., Formal Description of Programming Concepts, IFIP TC-2 Working Conf., Garmish-Partenkirchen, Fed. Rep. Germany, 1982 (North-Holland, Amsterdam, 1983) 199–223.

[12] J.-P. Queille and J. Sifakis, Specification and verification of concurrent systems in CESAR, in: Proc. 5th Internat. Symp. on Programming, Paris, 1981.

[13] D. Zöbel, Normal Form Transformations for Programs in CSP, Rept., EWH Rhld.-Pf., Abteilung Koblenz, Seminar für Informatik, 1987.