Modeling the Distributed Termination Convention of CSP

KRZYSZTOF R. APT
Université Paris 7
and
NISSIM FRANCEZ
Technion—Israel Institute of Technology

How the distributed termination convention of CSP repetitive commands can be modeled using other CSP constructs is shown. The presented transformation suggests a simple implementation of this convention. We argue that this convention should be used as a compiler option.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management

General Terms: Algorithms, Languages, Verification

Additional Keywords and Phrases: Distributed termination convention, CSP, program transformation, deadlock

1. INTRODUCTION

In 1978 Hoare [11] introduced CSP, a language for distributed programming. One of its features, which was subsequently criticized, was the so-called distributed termination convention of repetitive commands (see, e.g., [12]).

A repetitive command is a construct of the form

$$*\,[b_1;\ \alpha_1 \to S_i \;\square\; \vdots \;\square\; b_m;\ \alpha_m \to S_m],$$

where $b_1, \ldots, b_m$ are Boolean expressions, and $\alpha_1, \ldots, \alpha_m$ are input commands. Constructs $b_i;\ \alpha_i$ are called *guards*. (In fact, guards in CSP are of slightly more general form; for example, they contain variable declarations. Our result applies

Authors' addresses: K. R. Apt, LITP, Université Paris 7, 2, Place Jussieu, 75251 Paris, France; N. Francez, Computer Science Dept., Technion—Israel Institute of Technology, Haifa 32000, Israel.

to the general case as well.) Such a loop is said to terminate if all its guards *fail*. A guard b_i; α_i fails if either b_i evaluates to **false** or the process addressed in α_i has terminated. This convention of exiting a loop was subsequently called the *distributed termination convention* (DTC). Hoare himself expressed in [11] some worries about this convention and indicated in an example how the desired effect of a loop exit can be achieved using other primitives of the language.

In this paper we show how, in the case of arbitrary programs, the distributed termination convention can be modeled in an extension of the original CSP, in which output guards are allowed. The transformation we provide suggests a simple implementation of this convention. We argue that this convention should be used as a compiler option, as it is a powerful programming tool, relieving the programmer of a major concern.

In the original CSP, output guards were disallowed. Subsequently, this restriction has often been criticized (see, e.g., [6]). In various CSP programs given in the literature (see, e.g., [8]) this restriction is not used. Also, in a variety of semantic definitions and proof systems for the language (see, e.g., [10] and [13]), this restriction is disregarded. The cost of implementing "handshaking" turned out to be less inefficient than Hoare thought it might be (see [7] for a survey of such implementations and further references). In this paper we allow output guards, and consequently the proposed transformation deals with repetitive commands of a more general form where $\alpha_1, \ldots, \alpha_m$ are I/O commands. The proposed transformation shows that the distributed termination convention can be defined using other CSP language features *provided* output guards are allowed.

As a further motivation for an explicit definition of such a transformation, one could consider its bearing on other transformations related to CSP. As a concrete example, consider a recent work [5], where CSP is modeled using Milner's calculus of communicating processes (CCS) [14]. The DTC influenced the complexity of this modeling in that an extra process was created only for the purpose of centrally handling the termination information; this process could be avoided in view of the suggested transformation.

2. DISCUSSION

Let us start by recalling Hoare's example. Consider the following program, which represents a process computing integer division with remainder and a user process.

```
[DIV :: *[X?(x, y) →
          quot := 0; rem := x;
          *[rem ≥ y → rem := rem − y; quot := quot + 1];
          X!(quot, rem)]
 ‖ X :: USER
]
```

The DIV process leaves its main loop as soon as the user program terminates. This effect can be achieved by explicit exchange of **end** signals. The following rewritten version of the program is equivalent to the original version. According

where the variables continue$_Y$ and continue$_Z$ are originally intialized to **true**. These variables are also used to refine the repetitive commands of X in a manner analogous to the case of the DIV process.

If continue$_Y$ turns to **false** before the loop is reached, this is due to the termination of the process Y. But if Y has terminated, it is useless to send it the information about the termination of X and indeed such a sending will not occur here; similarly with the process Z. Thus continue$_Y$ now represents the process X's knowledge of the fact that the **end** signal has been received from Y *or* sent to Y. Thanks to this double role of the continue variables, we have guaranteed the termination of the loop in the cases in which the process X has received *or* sent **end** signals from (or to) both Y and Z.

Now what about a situation in which no exchange of the **end** signal takes place between X and, say, Y? Such a situation will arise when (the original versions of) X and Y have terminated without relying on each other's termination. In this case we would also like to force the termination of the loops, added at the end of X and Y, respectively. This can be achieved by arranging another exchange of **end** signals between the new versions of X and Y.

The loop added at the end of Y contains a guard with the communication X!**end**. This communication can now be executed if the loop at the end of X contains a guard with Y?**end**. After this communication both loops should terminate (in case of two processes).

Summarizing the loop added at the end of X should take the following form:

```
*[ continueY; Y!end  → continueY := false
 □ continueY; Y?end  → continueY := false
 □ continueZ; Z!end  → continueZ := false
 □ continueZ: Z?end  → continueZ := false].
```

Symmetric loops should be added at the end of Y and Z, respectively. In this solution we can drop some of the input guards in the above loops. For example, the process Y does not need to contain a guard with X?**end**, since the final communication between X and Y is already assured by the pair X!**end** and Y?**end**.

So far we have dealt with repetitive loops in which guards do not contain a Boolean part. It turns out, however, that the above solution can be straightforwardly generalized for the case of arbitrary repetitive commands. We now present this solution and prove its correctness.

3. A GENERAL SOLUTION

Consider a parallel command $P \equiv [P_1 :: S_1 \parallel \cdots \parallel P_n :: S_n]$, where $P_1, \ldots, P_n$ are processes with bodies $S_1, \ldots, S_n$, respectively. Assume that each P_i refers to some P_j's ($j \neq i$) only. In other words, no P_i refers to an external process. Let **end** be a communication signal of a new type.

For each P_i, let Γ_i be the collection of indices of all processes referred to by guards in some loop in P_i. In other words, $j \in \Gamma_i$ iff P_i contains some loop L, one of whose I/O guards refers to P_j. Note that by the CSP rules for each i, Γ_i can be syntactically determined. If $j \in \Gamma_i$, then we call P_j the *neighbor* of P_i. Note that the neighborship relation does not need to be symmetric: Γ_i and Γ_j for $i \neq j$ do not depend on each other.

to this version, a loop terminates as soon as all the Boolean parts of its guards evaluate to **false**.

```
[DIV :: continue := true;
  *[continue; X?end → continue := false
    □ continue; X? (x, y) →
      quot := 0; rem := x;
      *[rem ≥ y → rem := rem − y; quot := quot + 1];
    X!(quot, rem)]
‖ X :: USER; DIV!end
]
```

Note that the above solution works for a general case of two processes when one process P repeatedly leaves its loops due to termination of another process Q. Once the process P leaves a loop due to termination of Q, the variable continue is set to **false**. From that stage on, the variable continue remains **false**. This represents the process P's knowledge of the fact that Q has terminated. All future communications with Q in a guard of a repetitive command will now fail, as desired, since continue is included in the guard. The **end** signal will be sent from Q to P only once.

A problem arises if more than two parallel processes are considered. Suppose that the process X communicates with processes Y and Z. Adding at the end of X

Y!**end**; Z!**end**

poses a problem since Z can then receive the **end** signal from X (informing about the termination of X) only after Y has received it. An alternative solution

[Y!**end** → skip □ Z!**end** → skip]

does not work either, as now only one process among Y and Z can make use of the termination of X. We might also try adding at the end of X

[Y!**end** ‖ Z!**end**].

But now the new version of X can terminate *only* if both Y and Z have made use of the termination of (the old version of) X. It is easy to see that if the above transformation is applied in a symmetric manner to X, Y, and Z, then original properly terminating computations become deadlocked ones.

Note that an analogous problem already arises in the case of two processes P and Q when P and Q terminate without relying on each other's termination.

A similar drawback results if we add at the end of X a loop

*[Y!**end** → skip □ Z!**end** → skip].

An addition of a loop at the end of X seems, however, to be the only possible solution. The process X should send an unknown number (0, 1, or 2) of **end** signals, and only a loop can possibly model such behavior.

First, we would like to force the termination of such a loop in case **end** signals have been sent to both Y and Z. This is easy—we simply refine the above loop in the following manner:

```
*[ continue_Y; Y!end → continue_Y := false
□ continue_Z; Z!end → continue_Z := false]
```

We use the "big box" notation $\Box_{j\in\Gamma_i} b_j \to S_j$ as a shorthand notation for $b_{j_1} \to S_{j_1} \Box \cdots \Box b_{i_k} \to S_{j_k}$, in the case $\Gamma_i = \{j_i, \ldots, j_k\}$.

We now transform each P_i into another process by applying the following steps.

(1) At the beginning of P_i add the following program section:

continue$(1 \ldots n)$:boolean, j:integer; $j := 0$;
$*[j < n \to j := j + 1;$ continue(j) := **true**$]$.

Rename the variables in order to avoid variable clashes and to keep the processes disjoint.

(2) Given a repetitive command

$$*[\Box_{j=1,\ldots,k} b_j;\ P_i\$x_j \to T_j]$$

within a process P_i (where \$ stands for ! or ?), replace it by

$$*[\Box_{j=1,\ldots,k} b_j;\ \text{continue}(i_j);\ P_{i_j}\$x_j \to T_j$$
$$\Box_{j=1,\ldots,k} b_j;\ \text{continue}(i_j);\ P_{i_j}?\textbf{end} \to \text{continue}(i_j) := \textbf{false}].$$

(3) At the end of P_i add the following program section:

$$*[\Box_{j:i\in\Gamma_j} \text{continue}(j);\ P_j!\textbf{end} \to \text{continue}(j) := \textbf{false}$$
$$\Box_{j\in\Gamma_i} \text{continue}(j);\ P_j?\textbf{end} \to \text{continue}(j) := \textbf{false}].$$

Note that the first part of the above repetitive command ensures sending the **end** signal to all processes P_j having P_i as a neighbor, whereas the second part offers reception of the **end** signal from any process P_j being a neighbor of P_i. As indicated at the end of the previous section, some of these input guards can be deleted. In fact we can replace Γ_i above by $\Gamma_i \cap \{j : i \in \Gamma_j \to j < i\}$.

Thus if, for example, $\Gamma_X = \{Y, Z\}$, $\Gamma_Y = \{X\}$, and $\Gamma_Z = \{Y\}$, then the following I/O guards appear in the repetitive commands added at the end of the processes X, Y, and Z, respectively:

X: $Y!$, $Y?$, $Z?$
Y: $X!$, $Z!$, $X?$
Z: $X!$, $Y?$

and according to the improved version:

X: $Y!$, $Y?$, $Z?$
Y: $X!$, $Z!$
Z: $X!$, $Y?$

In what sense does the above transformation model the distributed termination convention? Denote the transformed version of P by P'. Assume that computations of P can make use of the DTC, whereas those of P' cannot.

When we say that a computation does not make use of the distributed termination convention, we simply mean that it does not rely on the fact that some of the processes have terminated. In the terminology of [8] this corresponds to the notion of endotermination and endoprocess. Therefore we call such

computations *endocomputations*. We have the following theorem:

THEOREM 1. *Every properly terminating computation of P can be extended to a properly terminating computation of P'.*

PROOF. Consider first an endocomputation of P. In such a computation all loops terminate owing to an eventual failure of all the Boolean components of the guards.

Consider now an extension of this computation obtained by executing first the parts added to $P_1, \ldots, P_n$ in step (1). Clearly, in this way we obtain a partial computation of P'. This computation reaches the parts added to $P_1, \ldots, P_n$ in Step (3) by the assumption of proper termination. We can now extend this partial computation by exchanging, one by one, the **end** signals between all pairs of processes that are neighbors. Given two processes P_i and P_j $(i \neq j)$ such that $i \in \Gamma_j$ or $j \in \Gamma_i$, the **end** signal will be exchanged between them exactly once. After having exchanged all **end** signals, all processes terminate. We thus obtain a properly terminating computation of P'.

Consider now a properly terminating computation of P, which makes use of the DTC.

Suppose that the convention is used for the first time when the loop

$$*[\underset{j=1,\ldots,k}{\square} b_j;\ P_i\$x_j \rightarrow T_j]$$

within some P_i is exited owing to the falsity of b_j for $j \notin A$ and the termination of P_{i_j} for $j \in A$, for some nonempty set $A \subset \{1, \ldots, k\}$. Here again \$ stands for ! or ?.

Now take the previously considered initial extension of the computation of P. At the moment of the above loop exit insert in it, for all $j \in A$, a communication of the **end** signal between P_i and P_{i_j} followed by setting the corresponding continue variables of P_i and P_{i_j} to **false**. Now the corresponding loop of P_i can be exited, since for each $j \in \{1, \ldots, k\}$ either b_j or continue(i_j) evaluates to **false**.

Similar additions deal with other loop exits due to the DTC. In cases in which the process P_i has already made use of the termination of some P_{i_j}, the variable continue(i_j) is already set to **false**, and no additional exchange of the **end** signal between P_i and P_{i_j} takes place.

In such a way we obtain a computation of P' that reaches the parts added in Step (3). We can now extend this computation to a terminating one by exchanging **end** signals between all pairs of neighbor processes that have not yet done so.

This concludes the proof. □

We also have a converse theorem.

THEOREM 2. *Every properly terminating computation of P' can be restricted to a properly terminating computation of P.*

PROOF. We can apply an inverse procedure to the one considered in the proof of Theorem 1. □

Analogous theorems hold for infinite computations of P and P'. Note, however, that the theorems do not hold for failing or deadlocked computations. (Recall

that a computation is failing if it reaches an alternative command with all guards failing.) Every deadlocked computation of P can be extended to a deadlocked computation of P'. The converse is not necessarily true. As an example consider the program

$$P \equiv [P_1 :: [P_2?x \rightarrow \text{skip}]; *[P_2?x \rightarrow \text{skip}] \parallel P_2 :: \text{skip}].$$

Then P' is of the form

$$P' \equiv [P_1 :: \text{initialization part}; [P_2?x \rightarrow \text{skip}]; \ldots \parallel P_2 :: \text{initialization part}; \text{skip}; \text{termination part}].$$

According to the semantics of CSP, the only computation of P is a failing one (the first guard $P_2?x$ of P_1 fails as soon as P_2 terminates), and all computations of P' are deadlocked ones.

Thus an extension of a failing computation of P can become a deadlocked computation of P'. Conversely, a restriction of a deadlocked computation of P' can become a failing computation of P. On the other hand, a restriction of a failing computation of P' becomes a failing computation of P.

We can obtain a one-to-one correspondence between failing and respectively deadlocked computations of P and P' if, in addition, we transform all I/O commands and alternative commands of P appropriately.

First, we replace any I/O command α within P by $[\alpha \rightarrow \text{skip}]$. Next, we replace any alternative command

$$[\underset{j=1,\ldots,k}{\square} b_j; P_{i_j}\$x_j \rightarrow T_j]$$

by the following program

more := **true**;
*[more →
$[\underset{j=1,\ldots,k}{\square} b_j; \text{continue}(i_j); P_{i_j}\$x_j \rightarrow T_j; \text{more} := \textbf{false}$
$\underset{j=1,\ldots,k}{\square} b_j; \text{continue}(i_j); P_{i_j}?\textbf{end} \rightarrow \text{continue}(i_j) := \textbf{false}]$.

Finally, in Step 3 of the previous transformation we should now use a new neighborhood relationship, which was before confined to loop-connectedness: $j \in \Gamma_i$ iff P_i contains *any* guard (not necessarily in a loop) referring to P_j.

The proof that the above transformation has the desired properties is similar to the proof of Theorem 1 and is left to the reader. Clearly the above transformation of the alternative commands does not affect correctness of the previous transformation modeling the distributed termination convention.

Note that the transformed alternative commands can lead to failure only owing to the falsity of all the Boolean parts of its guards. We can thus say that the above transformation of alternative commands models the CSP convention of a failure of alternative commands using a simpler convention, according to which an alternative command fails only when all the Boolean parts of its guards evaluate to **false**.

Finally, we would like to make the following observation. When some transformed process sends the **end** signal to another process, it *has not yet actually terminated.* It still has to terminate its termination part introduced by the transformation. Thus another process may exit a loop "somewhat earlier" than

prescribed by the original semantics, requiring that all processes referred to by loop I/O guards *have actually terminated.* However, the DTC convention was chosen [Hoare, private communication] with the following question in mind: "*When should an I/O guard be considered to be false*?" Any acceptable decision should imply that once a guard $P_j?x$ (or $P_j!x$) in P_i is false, it should be guaranteed that P_j *will never again attempt communication with* P_i. Hoare felt that the only plausible assurance is the absolute termination of P_j's. Our transformation provides a different interpretation to the falsity of an I/O guard. The addressed process still has to execute a **finite computation**, guaranteed not to attempt communication with a process to which the **end** signal has been sent. Indeed, the continue variables take care of this. This modification has no observable semantic difference from the original interpretation.

So far we have dealt only with programs disallowing nested parallelism. It is, however, straightforward to see that the above transformation also works in the case of arbitrary programs. Nested parallelism introduces scoping problems, which have now to be examined more closely. Consider for example a program P of the form

$$P \equiv [P_1 :: [P_{11} :: S_1 \parallel P_{12} :: S_2] \parallel P_2 :: S]$$

According to the scoping rules, the process P_{11} can refer to P_2 but not vice versa—P_2 can only refer to P_1. But this simply means that the neighborhood relationship between P_{11} and P_2 is not symmetric (in cases when P_{11} refers to P_2). No other complications arise here, and the proof of correctness of the above transformations is the same as before.

4. CONCLUSIONS

The modeling of the distributed termination convention presented above is both simple and efficient. No new communication channels are introduced. The number of additional communications is bounded by the number $K = \sum_{i=1}^{n} \Gamma_i/2$ and, in the case of properly terminating computations, equals this number. In the terminology of [8] we transformed each exoprocess into an equivalent endoprocess.

The transformation we have presented suggests how to implement the DTC. Moreover Theorems 1 and 2 prove the correctness of such an implementation. We can compare the situation with the use of recursion in procedural programming languages. Transformations that remove recursion using a stack suggest how recursion should be implemented. In both cases preprocessing of the original program by applying the corresponding transformation leads to inefficient implementation. Transformations should rather be used as a guideline for implementing the above features at a lower level.

In several programs the DTC is not needed to cause loop termination, and the usual convention of leaving a loop when all the Boolean parts of its guards evaluate to **false** already suffices. In such cases implementation of the DTC causes an unnecessary overhead.

This deficiency can be avoided by using the DTC as a *compiler option* similar to, for instance, the option **packed** in Pascal.

Only repetitive commands preceded by the keyword **distributed** would then be implemented by taking into account the DTC. (To keep in spirit with the

CSP compact notation, we might use "..." instead of the keyword **distributed**, thus writing ... S for repetitive commands S, which should be implemented with the distributed termination convention.) The other repetitive commands would be implemented by taking into account the above simpler convention.

The transformation from the previous section provides sufficient information on how to implement such an option. First, only repetitive commands preceded by the keyword **distributed** should be transformed, as indicated in Step (2) of the transformation. Second, the repetitive commands considered in Step (3) should refer only to the processes that in their program text contain the keyword **distributed**. They should be added at the end of the program text of the processes that are referred to in the guards of **distributed** repetitive commands.

Of course it would be then programmer's responsibility to decide properly which repetitive commands should be preceded by the keyword **distributed**.

The first of the proposed transformations can also be used to derive and justify a proof rule for repetitive commands dealing with the DTC, much in the same way as was done in [4] for the case of proof rules dealing with fairness. The transformation translates CSP programs into CSP programs that do not rely on the DTC. Since for the latter type of programs we have already a sound and relatively complete proof system (see [3] and [1]), by "absorbing" the transformation into the assertions we obtain a sound and relatively complete proof system for CSP programs in the case when the DTC is adopted. The corresponding proof rule for repetitive commands becomes after some simplifications essentially the rule provided in Section 4 of [3]. Exact details of this procedure are a bit tedious but completely straightforward.

Note. This paper originated from two reports [2] and [9] on the subject written by the authors independently.

ACKNOWLEDGMENT

The first author wishes to thank E.-R. Olderog for useful comments and discussion on the subject of the paper. The second author's initial attempts at the construction of the transformation were made with W. P. de Roever during his visit at the Computer Science Department of the Technion in Haifa.

REFERENCES

1. APT, K.R. Formal justification of a proof system for communicating sequential processes. *J. ACM 30*, 1 (Jan. 1983), 197–216.
2. APT, K.R. Modelling the distributed termination convention of CSP. Tech. Rep. 83-10, LITP, Paris, 1983.
3. APT, K.R., FRANCEZ, N., AND DE ROEVER, W.P. A proof system for communicating sequential processes. *ACM Trans. Prog. Lang. Syst. 2*, 3 (July 1980), 359–385.
4. APT, K.R., AND OLDEROG, E.-R. *Proof Rules and Transformations Dealing with Fairness.* Vol. 3, *Science of Computer Programming.* Elsevier North Holland, New York, 1983.
5. ASTESIANO, E., AND ZUCCA, E. Semantics by translation of CSP and its relationship with β-semantics. In *Proceedings of the 10th MFCS.* Lecture Notes in Computer Science, vol. 118. Springer-Verlag, New York, 1981.
6. BERNSTEIN, A. Output guards and nondeterminism in Communicating Sequential Processes. *ACM Trans. Prog. Lang. Syst. 2*, 2 (Apr. 1980), 234–238.
7. BUCKLEY, G.N., AND SILBERSCHATZ, A. An effective implementation for the generalized input-output construct of CSP. *ACM Trans. Prog. Lang. Syst. 5*, 2 (Apr. 1983), 223–235.

8. FRANCEZ, N. Distributed termination. *ACM Trans. Prog. Lang. Syst. 2*, 1 (Jan. 1980) 42–55.
9. FRANCEZ, N. Program transformations eliminating the distributed termination convention of CSP. Tech. Rep. RC-9935, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1983.
10. FRANCEZ, N., LEHMANN, D., AND PNEULI, A. Linear history semantics for distributed languages. In *Proceedings 21st FOCS Conference*, (Syracuse, N.Y., October) 1980.
11. HOARE, C.A.R. Communicating sequential processes, *Commun. ACM 21*, 8 (Aug. 1978), 666–677.
12. KIEBURTZ, R.B., AND SILBERSCHATZ, A. Comments on "Communicating sequential processes". *ACM Trans. Prog. Lang. Syst. 1*, 2 (Oct. 1979), 218–225.
13. LEVIN, G.M., AND GRIES, D. A proof technique for communicating sequential processes. *Acta Inf. 15*, 3 (1981), 281–302.
14. MILNER, R. *A Calculus for Communicating Systems.* Lecture Notes in Computer Science, vol. 92. Springer-Verlag, New York, 1980.

Received March 1983; revised December 1983; accepted January 1984