Retracing CSP

Stephen Brookes

Department of Computer Science
Carnegie Mellon University
Pittsburgh, USA

Abstract

The original CSP was a language for parallel imperative programs communicating by synchronized message-passing. Most of the early foundational work concerned a more abstract process algebra, now known as Theoretical CSP (or TCSP). The early semantic models involved communication traces, refusals, failures, and divergence traces. These models support compositional reasoning about safety properties, but since they do not assume fair parallel scheduling they are less well suited for proving liveness properties. More recent developments using a suitably formulated form of action traces provide a unifying semantic framework, applicable both to CSP-style synchronized communication and to asynchronously communicating processes, as well as to shared memory parallel programs, in each case assuming a simple form of fair execution.

Key words: concurrency, shared memory, communicating process, granularity, race condition, denotational semantics, logic

1 Background

The original CSP programming language [13], introduced by Tony Hoare in 1978, combined input and output with guarded commands [8] and parallel composition. For various practical reasons, the language imposed static syntactic constraints on program structure: no nested parallelism, direct process-to-process communication, and no shared variables. Some possible alternative design choices were considered, such as the use of output guards, and whether to assume synchronized or asynchronous message-passing. The original language allowed only input guards, and adopted synchronized communication.

The early foundational efforts dealt with a process calculus (Theoretical CSP, or TCSP), derived from CSP by abstracting away from state [11]. A TCSP process performs *events* belonging to an abstract *alphabet*, and parallel

[1] Email: brookes@cs.cmu.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

composition involves a form of interleaving in which concurrent processes must synchronize on the events belonging to the intersection of their alphabets. Hoare's first proposal for a denotational semantics of TCSP [12], involving prefix-closed sets of *communication traces*, was suitable only for simple safety properties, and ignored the possibility of deadlock. In the *failures* model [11], communication traces were augmented with *refusal sets* designed specifically to model deadlock: a process deadlocks when it refuses every event in its alphabet. The *failures-divergences* model [6] improved further by including information about *divergence*, characterized as the potential for "infinite internal chatter". The failures-divergences model treats divergence as a catastrophe, arguing that a potentially divergent process is useless; some later variants of this model take a more relaxed view of divergence. Subsequently Bill Roscoe also developed a failures-divergences semantics incorporating state [21], for the programming language *occam*, an (imperative) ancestor of the original CSP. Roscoe's book [22] contains a detailed and extensive account of the family of models belonging to the CSP school.

TCSP has enjoyed much success as a process algebra for specifying and proving correctness of communicating processes, and the failures-divergences model forms the basis for the model-checker FDR [9]. Milner's CCS [16,17], based on a more discriminating notion of program equivalence (bisimulation), has achieved similar success and wide applicability, with semantic models such as synchronization trees and labelled transition systems, and model-checking tools such as the Concurrency Workbench [7]. Both process algebras provide a succinct and expressive notation for specifying parallel systems together with algebraic laws of program equivalence. Indeed, a collection of CSP laws valid for failures-divergences semantics can be used to justify the normal form property for processes that is a key ingredient in implementing FDR, and the Concurrency Workbench builds on top of CCS laws expressing properties of bisimulation equivalence.

2 Reassessment

Now that more than 25 years have passed since the beginnings of CSP, it is worth looking back, with the benefits of hindsight and experience, reassessing some of the early design choices in the light of later developments and pointing out limitations that may have seemed unimportant at the time but warrant further investigation or reconsideration.

The early models of TCSP, and most of their successors, were concerned only with finite traces, and therefore did not (need to) assume any form of fair parallel execution. As a result, these models are not well suited for reasoning about liveness properties, such as the eventual inevitability of some desirable event: typically it is impossible to prove a liveness property without assuming that process execution and the use of shared resources is governed by a reasonably fair scheduler. At the time, fairness was regarded as semantically

problematic and difficult to incorporate into the denotational setting. David Park's classic paper [19] and later developments such as [5] showed how this could be done for shared memory parallel programs, but the notion of concurrency underlying CSP seemed radically different from the shared memory paradigm, and it was not easy to see how to combine CSP with fairness without requiring complicated book-keeping to keep track of scheduling information.

The early models of CSP also ignored the potential for race conditions, such as concurrent attempts to receive input from the same channel, or concurrent writes to the same variable. A program whose execution is susceptible to races may exhibit unpredictable behavior, and its safety and liveness properties may depend on implementation details beyond the control of the programmer. The syntactic constraints of the original CSP language obviously suffice to rule out racy programs, by banning shared variables and imposing limits on channel usage. However, these syntactic constraints seem unnecessarily draconian: it seems natural to allow nested parallel composition, and to allow processes to use a combination of shared state and channel-based communication. Furthermore, a similar approach cannot be adopted if we extend CSP with pointers and mutable state, since syntax-based analysis would then longer suffice to detect sharing. The TCSP models discussed above treat input and output as atomic actions, tantamount to assuming that the underlying implementation of a channel ensures that at most one process is allowed to input, and at most one process is allowed to output, at all stages. Again such assumptions obviate the need to deal semantically with racy behavior, but may not be realistic in practice.

All of the models mentioned so far were tailored specifically for modelling synchronized communication, and are not well suited for shared memory or asynchronous communication. Historically, these parallel paradigms have been endowed with separate families of semantic models, with origins in early work such as [10,15,16,11] and later more comprehensive accounts such as [14,17,22]. These families have disappointingly few structural similarities, a disparity that has tended to prevent semantically-based techniques for program analysis developed for one paradigm from being easily used in another. To an extent such differences are to be expected: in particular the CSP semantic models differ fundamentally from those developed for CCS, because traces, failures and refusals reflect a "linear time" view of process behavior whereas bisimulation fits the "branching time" view better. Yet there is much less reason to expect or require such disparity between models sharing the same linear-time view of behavior. None of these models is clearly "best", and such comparisons are fruitless: typically each applies to a limited class of programs, and deals with a different notion of program behavior. It seems natural to seek a single semantic framework capable of interpreting all of these paradigms as variations on a common theme.

3 Recent developments

Over the past few years we have developed a uniform family of semantic models, based on a form of *action trace*, suitable both for reasoning about shared memory parallel programs and about networks of communicating processes [4,2,1]. Furthermore, the framework is adaptable both for synchronized communication and for asynchronous communication. The framework can therefore be used to model a concurrent language that combines features from each of these paradigms, including shared memory, as well as traditional synchronization primitives such as semaphores and monitors. Indeed, the framework can also handle mutable state such as pointers [3].

We have shown how to incorporate an intuitively natural notion of fairness, so that our models are suitable for reasoning about safety and liveness properties. Unlike the earlier models, we no longer work with "partial" traces that represent prefixes of computations, and we do not augment trace sets with separate information such as refusal sets or divergences; instead we include "complete" traces, and employ a trace structure general enough to represent deadlock and divergence directly. We handle deadlock and divergence by means of idling steps, parameterized by a set of "directions" that indicate the reason for idling [4,2]. We do not equate divergence with disaster, since it seems quite straightforward to represent divergence as just another kind of trace: a divergent or deadlocked process performs an infinite sequence of idling steps. The use of complete traces, containing information about idling, is a key to handling fairness in a compositional manner.

Action trace models such as these can be shown to be grainless [2], i.e. independent of assumptions about the granularity of hardware operations and details such as word size; the key idea behind this achievement is a semantic characterization of race conditions and a definition of parallel composition that treats a potential race as a runtime error, following a suggestion of John Reynolds [20]. Our semantics can therefore be used to characterize those programs which are race-free from a given state, so that the model can be used to prove correctness properties together with a guarantee that execution is free from runtime errors and that program behavior is independent of granularity.

Action trace semantics makes appropriate distinctions between processes on the basis of their deadlock potential and their safety or liveness properties, and can therefore be seen as a generalization of the early CSP models [2], although we take a more liberal view of divergence. Our model is applicable to a rather more general language than the original [13], without the need to impose syntactic limitations [2].

We can identify laws of program equivalence specific to each concurrency paradigm, and laws whose validity relies crucially on fairness. Although we lack the space here to supply the semantic details, we will give a few character-

[2] Of course, for programs in the original CSP our semantics can be simplified by omitting race detection.

istic examples and some key laws. The reader should refer to the cited papers for the semantic definitions behind these laws. We write $[\![P]\!]$ for the trace set of process P, and we use juxtaposition of trace sets to denote concatenation. This trace semantics can be defined in the denotational style, by structural induction.

As a simple shared memory example, we have the following "expansion" theorem, when x and y are distinct identifiers:

$$[\![(x{:=}v_1; P)\|(y{:=}v_2; Q)]\!] = [\![x{:=}v_1]\!]\,[\![P\|(y{:=}v_2; Q)]\!] \;\cup\; [\![y{:=}v_2]\!]\,[\![(x{:=}v_1; P)\|Q]\!].$$

Furthermore we have $[\![(x{:=}v_1; P)\|(x{:=}v_2; Q)]\!] = [\![\mathbf{abort}]\!]$, since concurrent assignments to the same variable cause a race.

For synchronized communication we have

$$[\![\mathbf{local}\ a, b\ \mathbf{in}\ (a!0; b!0)\|(a?x; b?y)]\!] = [\![x{:=}0; y{:=}0]\!] = \{x{:=}0\, y{:=}0\}$$

$$[\![\mathbf{local}\ a, b\ \mathbf{in}\ (a!0; b!0)\|(b?y; a?x)]\!] = [\![\mathbf{while\ true\ do\ skip}]\!] = \{\delta^\omega\},$$

the second example illustrating how we model deadlock. We also have laws such as the following:

$$[\![\mathbf{local}\ h\ \mathbf{in}\ (h?x; P)\|(Q_1; Q_2)]\!] = [\![Q_1; \mathbf{local}\ h\ \mathbf{in}\ (h?x; P)\|Q_2]\!]$$

$$[\![\mathbf{local}\ h\ \mathbf{in}\ (h!v; P)\|(Q_1; Q_2)]\!] = [\![Q_1; \mathbf{local}\ h\ \mathbf{in}\ (h!v; P)\|Q_2]\!]$$

when h is not free in Q_1, and

$$[\![\mathbf{local}\ h\ \mathbf{in}\ (P_1; h?x; P_2)\|(Q_1; h!v; Q_2)]\!] = [\![(P_1\|Q_1); x{:=}v; \mathbf{local}\ h\ \mathbf{in}\ (P_2\|Q_2)]\!]$$

when h is not free in P_1 or Q_1.

These laws, expressing "inevitability" properties of code fragments in certain parallel contexts, rely on fairness for their validity.

For asynchronous communication we assume as usual that output to a channel is always enabled, but a process attempting input must wait if the channel is currently empty. We model a channel as a queue-valued variable. In contrast with the synchronous case we have

$$[\![\mathbf{local}\ a, b\ \mathbf{in}\ (a!0; b!0)\|(a?x; b?y)]\!] = [\![x{:=}0; y{:=}0]\!] = \{x{:=}0\, y{:=}0\}$$

$$[\![\mathbf{local}\ a, b\ \mathbf{in}\ (a!0; b!0)\|(b?y; a?x)]\!] = [\![y{:=}0; x{:=}0]\!] = \{y{:=}0\, x{:=}0\},$$

and because of race conditions involving concurrent input or output to the same channel we have $[\![(h!v_1; P)\|(h!v_2; Q)]\!] = [\![(h?x; P)\|(h?y; Q)]\!] = [\![\mathbf{abort}]\!]$.

Using the obvious list notation for queues, we have laws such as:

$$[\![\mathbf{local}\ h = \epsilon\ \mathbf{in}\ (h?x; P)\|(Q_1; Q_2)]\!] = [\![Q_1; \mathbf{local}\ h = \epsilon\ \mathbf{in}\ (h?x; P)\|Q_2]\!]$$

when h is not free in Q_1, as for the synchronous case; also

$$[\![\mathbf{local}\ h = L\ \mathbf{in}\ (h!v; P)\|Q]\!] = [\![\mathbf{local}\ h = enq(v, L)\ \mathbf{in}\ P\|Q]\!]$$

when $h!$ is not free in Q, and

$$[\![\mathbf{local}\ h = L\ \mathbf{in}\ (h?x; P)\|Q]\!] = [\![\mathbf{local}\ h = L'\ \mathbf{in}\ (x{:=}v; P)\|Q]\!]$$

when $deq(L) = (v, L')$ and $h?$ is not free in Q. We also have

$$[\![\textbf{local } h = \epsilon, k = \epsilon \textbf{ in } (P_1; h?x; k!\_; P_2) \| (Q_1; h!v; k?\_; Q_2)]\!]$$
$$= [\![(P_1 \| Q_1); \textbf{local } h = \epsilon, k = \epsilon \textbf{ in } x{:=}v; (P_2 \| Q_2)]\!]$$

when h, k do not occur free in P_1 or Q_1.

Again these laws embody fairness assumptions in a natural manner, allowing us to reason about a parallel system by assuming "without loss of generality" that some particular activity goes first.

Such laws can be extremely useful in calculational reasoning. These laws can be seen as ancestors of Milner-style expansion theorems [16,17] and the CSP laws presented in the early papers [11], but expressed in terms of a parallel programming language that stands as a true descendant of original CSP: an imperative concurrent language rich enough to encompass shared state, synchronous and asynchronous message-passing, nested uses of parallel composition, and a more flexible scoping mechanism for local data.

References

[1] S. Brookes. *A grainless semantics for parallel programs with shared mutable data.* Proc. Mathematical Foundations of Programming Semantics, Birmingham, England. May 2005. (Preliminary version.) Final version to appear, Elsevier ENTCS (2005).

[2] S. Brookes. *Retracing the semantics of CSP.* Invited paper, Proc. *25 Years of CSP* Conference, London, July 2004. In: **25 Years of CSP**, Springer LNCS Festschrift series, vol. 3525. Ali Abdallah, Cliff Jones, and Jeff Sanders, eds., 2005.

[3] S. Brookes. *A semantics for concurrent separation logic.* Invited paper, Proc. CONCUR 2004, London. Springer LNCS vol. 3170. August 2004.

[4] S. Brookes. *Traces, pomsets, fairness and full abstraction for communicating processes.* Proc. CONCUR 2002, Brno. Springer LNCS vol. 2421, pp. 466-482. August 2002.

[5] S. Brookes. *Full abstraction for a shared-variable parallel language.* Proc. 8th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press (1993), 98–109. Journal version in: *Inf. Comp.*, vol 127(2):145-163, Academic Press, June 1996.

[6] S. Brookes and A.W. Roscoe. *An improved failures model for CSP.* Proc. Seminar on concurrency, Springer-Verlag, LNCS 197, 1984.

[7] R. Cleaveland, J. Parrow, and B. Steffen. *The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems.* ACM TOPLAS, vol. 15, no. 1, January 1993, pp. 36–72.

[8] E. W. Dijkstra. *Cooperating sequential processes*. In: **Programming Languages**, F. Genuys (editor), pp. 43-112. Academic Press, 1968.

[9] "Formal Systems (Europe) Ltd." *Failures-Divergence Refinement, User Manual*. 1997.

[10] M. Hennessy, and G. Plotkin. *Full Abstraction for a Simple Parallel Language*, Proc. Mathematical Foundations of Computer Science Conference, Springer LNCS vol. 74, 1979.

[11] C.A.R. Hoare, S. Brookes and A.W. Roscoe. *A Theory of Communicating Sequential Processes*, J. ACM, July 1984.

[12] C. A. R. Hoare. *A model for communicating sequential processes*. In **On the Construction of Programs**, R. M. McKeag and A. M. MacNaughten, eds, pp. 229-254. Cambridge University Press, 1980.

[13] C.A.R. Hoare. *Communicating Sequential Processes*, Comm. ACM, 21(8):666–677, 1978.

[14] C.A.R. Hoare. **Communicating sequential processes**. Prentice Hall, 1985.

[15] G. Kahn and D.B. MacQueen. *Coroutines and Networks of Parallel Processes*, Proc. Information Processing '77, North Holland, 1977.

[16] R. Milner. *A Calculus for Communicating Systems*. Springer LNCS, vol. 92 (1980).

[17] R. Milner. **Communication and concurrency**. Prentice Hall, 1989.

[18] S. Owicki and L. Lamport. *Proving liveness properties of concurrent programs*, ACM TOPLAS, 4(3): 455-495, July 1982.

[19] D. Park. *On the semantics of fair parallelism*. In: **Abstract Software Specifications**, Springer-Verlag LNCS vol. 86, 504–526, 1979.

[20] J. C. Reynolds. *Towards a grainless semantics for shared-variable concurrency*. Invited Lecture, Proc. 31^{st} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Venice. ACM Press, January 2004.

[21] A. W. Roscoe. *Denotational semantics for occam*. In: Seminar on concurrency, Springer LNCS 197 (1985), pp. 306-329.

[22] A. W. Roscoe. **The Theory and Practice of Concurrency**, Prentice Hall, 1998.