The transputer and パーソナル

C.A.R. Hoare

訳者まえがき

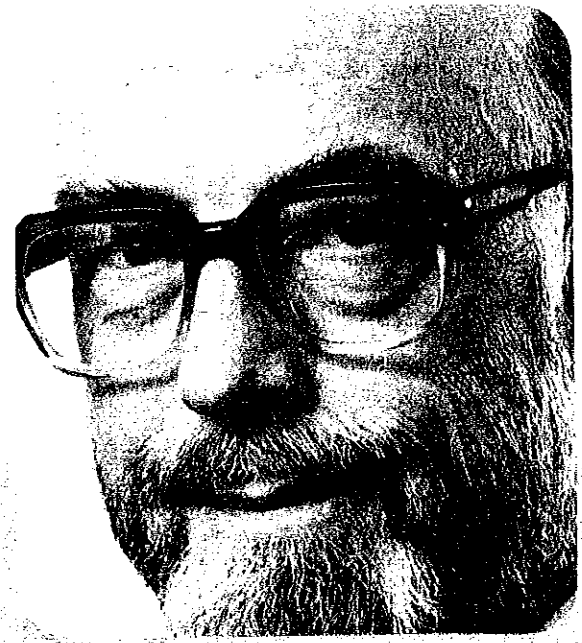
CSP理論，occam，Transputerといった一連の設計理念はコンピュータサイエンスだけでなく工学の分野においてもまれにみる良いお手本であると思います．このエレガントな手法のマジックはOxford大学における数学的理論が基本となっています．さらにHoare教授の一貫した指導力はアカデミックな領域だけでなく，産業界(Inmos, Formal Systems, etc) にも影響を及ぼしている事実は驚くべきことです．この論文はTransputer，occam開発の経緯だけでなくコンピュータサイエンスのエピソードについても詳しく述べられており，興味深い話題が数多く見られます．多くの読者の方々がTransputer，occamの隠れた並列処理における数多くのすばらしい事実を発見されることを期待致します．

要　約

この論文は25年間にわたる私のCSP (Comunicating Sequential Process) のアイディアの開発経緯について述べてみました．最も巧みで，有用な機能として“Guard”による選択があります．これはoccamプログラムのALT命令として使われています．この巧みさ，明快さ，さらに有用さというのは，Oxford大学により指導された数学的研究に基づく代数的法則を理解することによって益々盛んになります．数学に基づく抽象的概念はoccamの隠れた才能であり，それはマルチプログラミング，マルチプロセッシング，またはハードウェアによって信頼性高く，効率良く実現されています．結論として，occamプログラミングのパラダイムは将来汎用並列マルチプロセッシングにとって最も効率良く，信頼性の高いものとして残るであろうと考えられます．

C.A.R. Hoare: The transputer and occam: a personal story, *CONCURRENCY: PRACTICE AND EXPERIENCE*, Vol. 3, No. 4, pp. 249-264, 1991.

occam: ストーリー



訳 松井和人

■

多くのことをお話しするのですが，まず約25年前の1965年から始めたいと思います．ディジタルコンピュータはその頃すでに25年間存在していたので，いまからみるとコンピュータの歴史は約50年になります．1965年は Oxford 大学 Programming Research Group が Christopher Strachey 氏によって設立されたときであり，また，最初の Queen's Award が産業界に贈られたときでもありました．さらに重要なことは，多くの人々がこの頃に生まれたということであります．

ちょうどその頃，私は小さな英国コンピュータメーカ Elliott Brothers で働いていました．私は，32 Kbyte メモリで実行速度が1秒間に数千命令のマシン Elliott 803 のための Algol 60 のコンパイラを完成しました．そしてこれは高々29,000 ポンドで売られていました．このマシンにはオペレーティングシステムはなかったものの世界中で2000人以上の顧客があり，英国において最も成功したコンピュータの一つでありました．そのとき，人々は人間の計算力よりも1万倍速いものを探し求めていました．

人間がいかに大きな数字であるかを知るには単に万の桁を置き換えるだけで簡単にできます．ここで旅行と速度の類似性を思い出してみましょう．人間は徒歩で1時間に約5マイルの速度で歩くことができます．車は人間の約10倍の速度をもち，飛行機は数百倍の速度をもっています．人間が数千倍の速度をもって旅行するにはスペース・シャトルに乗ることが必要でしょう．1万倍というのはスペース・シャトルと歩いている人間でなく，“はいはい”している赤ん坊を分離している要素であります．同様にこの倍率は，人間の計算力と最初のコンピュータ上で私がプログラミングをしたスピードの比較と同じスケールでありました．

しかし1965年までには，そのコンピュータは明らかに十分な速度ではありませんでした．米国 CDC 社はすでに第1世代のスーパーコンピュータ 6600 を作っていました．そして小さな英国の会社は，CDC 社の性能にマッチするための必要な技術を投資することが見込めませんでした．したがって，あるディレクターが考えた案として――“どうしたら小さくて安いコンピュータを多く結合し，大きくて速い性能をもつコンピュータができるだろうか？”――，私たちはこのアイディアを調査するために，各専門分野協力のタスクフォースを設定しました．ソフトウェア開発責任者としての私のレポートはきわめて非積極的でありました．というのは，私はプログラムを組む方法が見えなかったからです．しかし，いま以上にその頃は，ソフトウェアの問題はその問題が必要に迫るまで無視されるのが通常であったようです．さらに当時としては，技術者はどうやってそのようなコンピュータを設計するのかを知らなかったし，またセールスマンもどうやってそれを売るのかを知らなかった時代でありました．したがって，私の同僚たちすべてのレポートが非積極的であるのは至極当然であったと思います．

会社はこのとき，通常のマシンのように多様性をも

つ大型で高速なコンピュータの設計に注目しました．このようなコンピュータは，もはやオペレーティングシステムと呼ばれるソフトウェアなしでは長く売れないことは確かでした．私たちはそれがどのようなオペレーティングシステムなのかは知っていました．しかし，どのようにして作るのかはわかりませんでした．私たちの試みは理論だけに留まり，失敗したことは事実でした．

私たちはその当時設計されていた他のいくつかのオペレーティングシステムを検討しました．そして可能性のあるモデルは，Sidney Michaelson のもとで設計されていた“Edinburgh Multiple Access System”でした．このシステムは独立したプロセスの集合から構成されており，ソフトウェアのシミュレーションによるメッセージの交換がなされていました．この方法は，オペレーティングシステムの問題を分けて解決することができました．そして私は，この方法がちょうど Algol 60 のコンパイラ設計においてトップダウン方式で再帰法を採用することでコンパイラが非常にシンプルになったように，システム全体を簡素化できると思いました．しかしこのとき会社は二つの大きな会社に吸収合併され，このプロジェクトは中止となり，私の推測は決してテストされることはありませんでした．

オペレーティングシステムの中に見られるプロセスには，ファイルの入出力とそれぞれ外部デバイスをスムーズにするためのバッファとかスプーラがありました．バッファは出力を実行するのにいつも Ready の状態をとり（バッファは空ではない），また入力を実行する際にもいつも Ready の状態をとります（いっぱいの場合を除く）．これらの動作に関する選択は，バッファリングのプロセスによって自律的にされなければなりません．しかし，これはユーザプログラムと外部デバイスの物理的ハードウェアのマシンに必要なタイミングにかなり依存することになります．したがって，私はこの概念を表現するプログラム言語を設計しました．この概念を occam で表現すると

```
ALT
  count> 0 & source ? buff[in]
    SEQ
      in := (in+1) \ size
      count := count-1
  count < size & request ? ANY
    SEQ
      sink ! buff[out]
      out := (out+1) \ size
      count := count+1
```

私は自信をもってこのアイディアを Edsger W. Dijkstra 氏へ手紙を書いたところ，彼の応答は失望的でありました．彼は，表記法と証明されていない複雑なシンタクスが必要とされるいかなる概念も嫌いなようでした．それはともかく，もしハードウェアデバイスとユーザプログラムの両方が同時に Ready の状態であればどうなるであろうか？ それはどちらが先に選択されて実行されるかは決定されない．1960 年代に私はまだ非決定性を恐れており，プログラミング言語の中に入れることは嫌いでした．いまになって再びその頃のアイディアを振り返ることができます．

私は，ALT コンストラクションはインタラクティブなプログラミング言語とリアルタイム・アプリケーションに対する occam の主な利点であると信じています．さらに，一般に並列プログラムにおける応答性と効率を増加させる方法をも提供できます．なぜそうなのか説明してみましょう．インタラクティブなコンピュータシステムはほぼすべての時間を待ち時間に費やさなければならないので，システムはその環境に対してアクションをとるのにつねに Ready の状態を保ちます．さもなければ，システムはコンピュータのアクションに対する待ち時間を多く費やす環境になります．これは不可能であり，耐えられないことでもあり，少なくとも好ましくないことです．そしてもっと安いコンピュータの出現によって，これは不要となっていくでしょう．しかし，待ち時間は決して自由時間ではありません．これをコンピュータで短くすることによって，ネットワーク全体に対するリソースの使用やシステム全体の効率を高めることができます．

しかし，どのようにして待ち時間を短くするのでしょうか？ 明らかな答えは WAIT 時間を速くすることです．現在のエレクトロニクスの進歩によれば，次世代のマイクロプロセッサは 10 倍以上の WAIT 時間の縮小が可能であります．そして，その効率は WAIT 時間に費やされる時間で制限されるように思われます．悲しいかな実はそうではありません．プロセッサのスピードが増加するにつれて，相対的な

WAIT 時間の非効率性は実際比例して増加します．したがって，私たちは解決を与えるソフトウェアの進歩に注目しなければならないのです．

実生活からこの類似点について考えてみましょう．ある通勤者が毎日会社をランダムに 5：30～6：30 の間に退社し，14 番のバスに乗り家へ帰ります．バスは 20～40 分ごとに到着するとします．したがって通勤者のバス停留所での平均待ち時間は約 30 分になります．もし通勤時間がたった 10 分だとすれば，この WAIT 時間は大変な遅れです．バスが 10 倍速く運転されてもそんなに早く家に到着することはできません．事実，バスの遅れは不満足でありましょう．WAIT 時間を短縮すればさらに良い結果となります．

しかし，必要となる効果は次のようにして得られます．通勤者が 14 番と同じ到着パターンをもつ 2 台目のバスのルート 2 番（会社と家の間を往復する）を知っているならば，これは同じ効果があります．ここで通勤者は会社から帰るとき，2 番か 14 番のどちらかのバスを待つことにします（どちらが先に来るかわからない）．こうすると，確率的には平均の待ち時間は半分になります（30 分から 15 分になる）．これは occam の ALT コンストラクションの秘密と同じです．

もちろん，これにはペナルティがあります．客はバスの選択のコントロールができないことです．もし 2 台のうち 1 台がときどき酔っ払いの群衆に乗っ取られるなり，また残りの 1 台が交通渋滞の中に入ったりするとき，客はもはや 2 台のうちどちらが乗り心地が良いか選択の余地がありません．バスに乗ってしまうとバスを変更するには遅すぎます．コンピュータの言語にはこの非決定的 (non-deterministic) な結果があります．そして，これにはバックトラッキング (back-tracking) がありません．occam における ALT 選択にも同じことが言えます．

1966 年，O. -J. Dahl と K. Nygaard 氏に招待され Oslo に参り，世界ではじめての目的指向型言語 Simula 67 のデザインのレビューを行ないました．この言語のオブジェクトは本質的におのおののローカルワークスペースであり，レジスタでは逐次プロセスであります．リモートプロシージャコールまたは直接調査 (direct inspection) する方法で通信されたオブジェクトは，ローカルなワークスペースを更新することができます．両方の繰返しは厳格ではあるが，優雅で階層化されたスコープ (scope) の制御によってコントロールされていました．この言語はもともとディスクリート・イベント・シミュレーション用に設計され，プロセスは疑似並列へと発展し，シミュレーション時間の経過によってコントロールされていました．この言語仕様は Algol 60 の拡張版でありました．

1968 年，私は Belfast にある Queen's 大学のコンピュータサイエンス学科の教授として移り，Simula 67 の概念の研究を引き続き行なう機会を得ました．これは後に，一連の出版物として発行することになりました．最初は Simula Classes[2]のための証明理論，後にオペレーティングシステム[3),4)]への応用として扱われました．すべてのアイディアは Jim Welsh と Dave Bustard によってに結びつけられました．そしてコンパクトに作られたのが，完全なオペレーティングシステムのモデルでありました．この結果は Structured System Programming[5]に報告されています．

1975 年，私は Marktoberdorf[6]のインターナショナル夏期講座でオペレーティングシステムの構造について講演しました．他の講演の中に E. W. Dijkstra の話がありました．彼は私のアイディアに感銘したが，再び表記法に強く反対しました．彼は，Algol 60 とか Simula 67 のようなプロシージャコールに関して説明された言語の意味の法則について強く反対しました．事実，法則は最も複雑な効果をもっており，ユーザプログラムの層はオペレーティングシステムの層との間で交錯していました．この問題を知的に解くのは，玉葱を中から外に剝くようなものでありました．

学生との夏期講座が Bavaria の山で行なわれたとき，Edsger と私はホテル Sepp の庭のテーブルにすわり，セマンティクスの記述と表記法の改良について論じました．これが CSP (Communicating Sequential Process) 最初のアイディアでした．これが出版されるまで，さらに 2 年の歳月を必要としました．この研究の目的は入力，出力，並列の組合せが多くの顕著な問題（プログラムの構造とプログラミング言語の設計において）を解決できることを示すことであり，複雑な特徴を置き換えたり，言語設計の仕様を大体達成することができました．

CSP の設計において，マイクロプロセッサの結合

されたネットワークの効率の良いインプリメンテーションを見ることが楽しみでした．ここで文献7)から引用してみます．

> *マイクロプロセッサの技術的進歩から同一のプロセッサを結合することによって，計算能力，容量，信頼性が増し，すべての物理的な並列処理がハードウェアによって隠されている一台のマシンに比べてさらに経済的である．ちょうどCDC 6600のマルチプルファンクション・ユニットにあるソフトウェアとか入出力コントロールパッケージのように…．*

これは大変退屈な予測でありました．なぜなら，この当時マルチプロセッサの試みは成功せず，値段も高く信頼性も低いものだったからです．

1977年，私は Oxford 大学のコンピュータ学科の教授として迎えられました．私はここ Oxford の PRG (Programming Research Group) において，Christopher Strachey および Dana Scott によって開拓されたプログラミング言語の設計と記述の数学的手法について何か学びとれることを期待しました．私は当時，学生であった John Kennaway, Steve Brookes, Bill Roscoe[8),9)]らを正しく指導することができました．彼らは後に CSP の標準的な数学的モデルを築きました．

同時に，Prentice Hall 社の Henry Hirschberg よりコンピュータサイエンス・シリーズの新しい企画について話がありました．表紙が赤と白でおなじみの本は，いまや50冊以上出版されています．Ian Barron は Inmos 社が設立されるときに，入力/出力チャネルをもち任意のサイズのネットワークの結合ができるマイクロプロセッサのデザイン・コンサルタントとして指導するように依頼に来られました．私は興奮を覚えて，自分の理論が David May と彼のソフトウェア，ハードウェアグループのチームによってどのように開発されるのか楽しみでした．そして5年後に彼らは遂に具体化し，Transputer という商品名でマーケットに出しました．Inmos 社の私の同僚たちにとっても，この時期は大変興奮に満ちた日々でありました．というのは，会社の将来のすべてはこの概念にかかっていたからであります．彼らは CSP, occam が正しいと信じて，いまでも会社に残って開発を続けています．

Transputer は，特に組込み型リアルタイムシステム用に設計されたプロセッサであります．したがってすばらしい入力/出力の機能をもち，信じられないほどの割込みへの応答性があります．これらの特徴は，ハードウェアに組み込まれているタイムシェアリング・オペレーティングシステム（マルチプログラミング）によってサポートされています．アプリケーションプログラムは並列のモデルを作ることにより，粗結合の並列プロセスの集合として自然に構成されます．マシンの低レベルのアセンブリ言語（後に occam と呼ばれる）は，ユーザが並列の制御を行なうために設計されました．そして並列プログラミングの難しさと危険性をコンパイルの方法で削減しようとしました．

Transputer は速く動作するように設計され，はじめて世に出たときは市場で最も速いマイクロプロセッサでありました．しかも本当に驚くのは，アプリケーションがさらに速いスピードを要求するときです．プログラムのプロセスを一つ以上の Transputer 上に分散することが可能であり，スピードを上げたければ単にチップを追加購入すればよいのです．どのプロセッサで実行させるかはプログラムを書いた後（アプリケーションの性能がテストされるとき）で決定できます．この利点は，リアルタイム・アプリケーションの難しいプログラミングを開放することを約束します．昔64 Kbyte メモリの容量の中にプログラムを詰め込んでいたときの問題点を，いまや誰が知っているでしょうか．いまや大きなサイズのメモリを購入することができます．

私がここで述べたことはすでに当たり前のことですから，たぶん許していただけると思います．ただ一つ学んだ基本的な原理はこういうことであります．プロセスが Transputer のネットワークに分散された場合の実行効果は，論理的に一個の Transputer 上でマルチプログラミングするのと同じであることを保証します．ただ一つの違いは（一般的に証明されている）実行速度です．この保証は，マルチコンピュータがもっと広く使われるときに必須なことです．これは Transputer が市場で生き残るのと，それを製造している会社にとって重大なことであります．

そのような保証を与える唯一の方法は，プロセスとまったく異なった二つの（あるいはもっと）方式，すなわち一対のワイヤによって物理的に分離されたプロ

セッサが通信するか，あるいは1個のプロセッサの中でタイムシェアリングすることによるどちらかの方法で，高い効率をもって実現できる理論的な通信の明確な概念をもったハードウェアの設計に基づくべきです．必要な概念は決してインプリメンテーションに関して記述できるのではなく，あらゆる場合の実現方法よりも，むしろより簡単でさらに抽象的な数学的モデルによって記述できます．

教訓的な話として，技術的かつ商業的に成功したTransputer は，Oxford で同時に開発された正当な数学理論の関連性に直接依存しています．私は，Transputer のすべてのユーザ（特に occam でプログラムを組んでいる人たち）はこういった背景を知らなくても，モデルの構成に役だった数学的研究に恩恵を受けていると思っていただきたいと思います．事実，たとえ製品のユーザがこのことを知らなくても（すべての工学分野であることだが），ユーザは決して技術者が設計の改良に使った技術を知る必要はありません．明らかに技術は製品の中に隠れているからです．

アプリケーションプログラム，汎用ソフトウェアのシステム，Transputer 自身のアーキテクチャを決定するために CSP モデルが適用されました．それは，ハードウェアの部品設計などにおいて詳細な仕様を記述する手段としての価値があることを証明しました．しかし，モデルの価値は数学的な特性，定義の理論的結果，それから動きを支配する法則などを研究することによってさらに増大します．そのようなコンピュータサイエンスの基本的な研究は単に長い報告書であり，またすべてギリシャ文字を使ったり，特殊文字を括弧と一緒に使ったりしているが，その内容は周知の事柄だったりすることがよくあります．実用的なプログラマかどうかを一日で見つけることが目的であっても，そのような論文は専門家が解読するにしても手に負えない仕事でありました．

しかし，プログラミングの法則に関する研究は異なります．その結果は，学校で学んだ代数と同じ代数の式として表わされます．各法則は別々に表現され理解されます．また，形式的な（formal）数学的証明から独立した自明の特質をもっているように思えます．これは，この法則が私たちの演算上の直観とプログラムの実行方法と非常に近いからであります．事実，法則の集合体はプログラミング言語の直観的理解を増強

させることができます．同様に，代数計算の法則は算術の理解に貢献しています．それではいくつかの例を使って説明しましょう．

最初の法則は並列に実行されるプロセスの例で，順序は問題ではありません．

```
PAR  =  PAR
  P       Q
  Q       P
```

これは交換の法則として知られており，同様に数の加算における交換の法則は

```
x+y = y+x
```

同様に，この法則は ALT コンストラクトにも適応できるが，もちろん連続ではありません．CSP の中では挿入辞 (infix) を使って並列の記述をします．

```
P || Q = Q || P
```

この記述は CSP の法則を短く表現でき，そしてたぶん覚えやすく，解かりやすく，応用もききます．しかし occam の数学的記述を使うことにします．なぜなら，すでに多くの方々がよくご存知だからです．

次の法則は並列プロセス間の通信を表わすからであります．

```
PAR   =   x:= e
  b ! e
  b ? x
```

この法則は，同じチャネルを介し入力 (b ? x) と出力 (b ! e) が並列で実行されることを表わします．最終的には，入力プロセスに割りつけられた変数 x に，出力プロセスに送られたある値 e が代入されることになります．これ以外に簡潔に表現できる方法はあるでしょうか？

いまここに並列プロセスがあり，二つのプロセスがチャネル b, c で接続され，そしてさらに，両方のチャネルのうち一方の入力だけが Ready となる ALT コンストラクトから成り立っているプロセスがあるとしましょう．次の例は，二つの選択を決定するのは出力プロセスであることを示します．すなわち，入力チャネルが Ready となっているほうであります．

```
CHAN b, c  =  x := e
PAR
  b ! e
  ALT
    b ? x
      SKIP
    c ? y
      SKIP
```

ALT の交換の法則によれば，もし出力プロセスがチャネル b のかわりに c に出力するようにした場合，その効果は x:= e から y := e へと変わります．

次に出力プロセスが選ばれていても，入力プロセスがチャネル c の入力に与えられない場合はどうなるでしょうか？　それは deadlock です！

```
CHAN b,c  =  STOP
  PAR
    b ! e
    c ? x
```

Deadlock は明らかに望ましくない現象です．ですから，私たちの理論は明確に定義されなければならないのです．題して “reason about it”．そして私たちが書いたプログラムで deadlock が起こらないことを証明しなくてはなりません．さもなければ，もっとやさしくて deadlock の危険性もない逐次プログラムを好んで書くようになるだろうと思います．

次の法則は並列 (PAR) と選択 (ALT) の関係を示します．入力を遂行する二つのプロセスが Ready であるとき，最初に実行されるのはプロセスの環境によって決定され，次に他の入力プロセスが後で実行されます．

```
PAR   =  ALT
  b ? x      b ? x
  c ? y      c ? y
           c ? y
           b ? x
```

もちろん，両方の入力が同時に起こることがあるかもしれませんが，私たちの数学的理論は論理的な特別な場合のように同時性について考える必要はありません．一つずつ順番が変わっても，それはいつも同じ論理的効果をもっているからです．

この法則は，有限な occam プログラムから並列のシステマティックな簡素化を行なった代表的な法則の例であります．occam における並列が簡素化できる

事実は，理論家にとって大いに励ましになります．これを見て，並列について神秘的でないと思わない人は誰もいないと思います．実際，プログラマにとって並列プロセスが一つのプロセッサ上で実行されているとき，その効率を上げるのは有益なことです．プログラム設計者にとって，異なったアーキテクチャ上で最適化されたり，あるいは異なった技術によるところの並列の構成がはなはだしく異なっている二つのプログラムの等価性を証明することは大変有益であります．最後に，Deadlock の存在を検出することも大事です．もし相当する逐次プログラムに STOP があればたぶん Deadlock だろうし，もしなければ並列バージョンも Deadlock はないことでしょう．

次の法則は，ALT コンストラクションがいかにこのような効果を与えるかを示した簡単な例です．これは逐次プログラマにとってはあまり知られていない非決定性の例です．

```
PAR   =  OR
  b ! e        x := e
  c ! f        x := f
  ALT
    b ? x
     c ? x
    c ? x
     b ? x
```

左側の 3 番目のプロセスは両方のチャネル b, c から入力待ちの選択であります (ALT)．これに対応する出力は他の並列プロセスで実行されます．しかし，ハードウェア中の occam の並列プロセスの相対的実行速度は正確に制御できません．したがって，二つの出力のうちどちらが最初に Ready になるかはわかりません．もし出力 b が最初だとすると x 値は f になり，もし出力 c が先に実行されると最終値 x は e になります．もし二つとも同時に Ready になれば，どちらが先に選択されるかは不明です．この現象を書くために，新しいコンストラクションでどちらか一方のプロセスが実行されることを示す OR を使っています．しかし，どちらが実行されるかは表現できません．この非決定的 OR は明らかに ALT と違っています．プロセスの選択は完璧に自律的で，しかも任意です．さらに環境によって，影響または制御できません．OR と ALT の明らかな差は数学的理論の成果です．

非決定性は，選択 (ALT) を使うことによって効率を上げようとする結果から生じます．しかし，論理的に望むべきプログラムの特徴ではありません．プログラムが正確になることは難しくなります．なぜなら，一つだけでなく両方の ALT が正しいことと，目的にかなった動きをすることを確かめなくてはならないからです．したがって，非決定性 OR コンストラクションは occam の中に含まれていません．この使い方では二つの正しいプログラムを書く必要性が生じ，まったく役に立ちません．しかし，非決定性をプログラムの理由として代数の中に含まれることは必須であります．なぜなら，それが非決定性を理解したり，制御したり，ハーネスを作ったりする唯一の方法だからです．

非決定性の一つの特別なハーネスは平凡な場合で，両方のコンポーネントは同じ論理的な動作をします．

```
OR    =   P
  P
  P
```

これは吸収 (idempotent) の法則として知られています．ブール代数における接続詞 (conjunction)，離接接続詞 (disjunction)，または二つの数の大小によって共有されます．

```
P /\ P = P
```

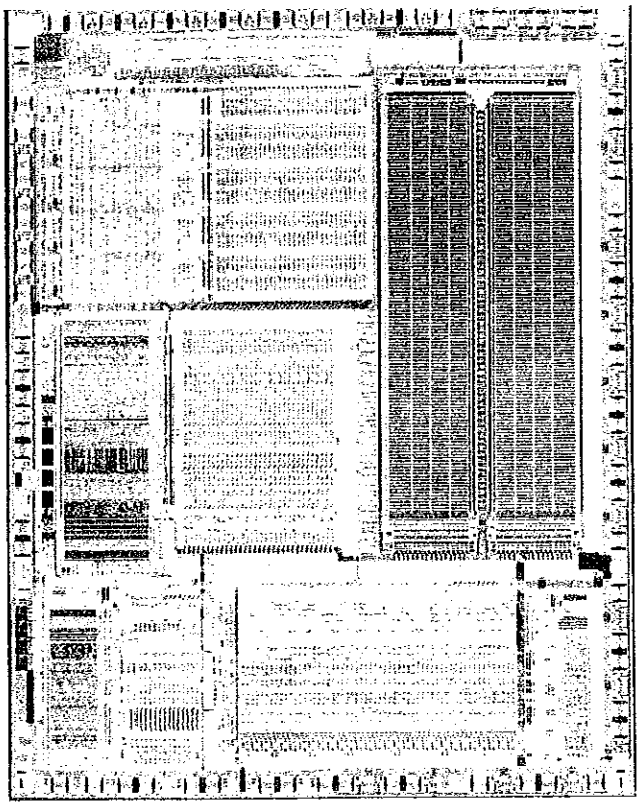
次の法則は非決定性の連続性を調べることと，期待しないものを取り除くのに有効です．

```
PAR      =   OR
  OR           PAR
    P            P
    Q            R
  R            PAR
                 Q
                 R
```

これは，並列が非決定性を通じて分配されることを表わします．同様に，算術の乗算は加算を通じて分配されます．

```
(p + q)*r = p*r + q*r
```

この法則は，いかなる算術式においても“+”を含



まない項の数を減じるので重要です．occam において WHILE を除き，各コンストラクションは非決定性を通して分配されます．有限なプログラムは非決定的なプロセスのリスト（おのおののプロセスは決定的である）として表現できることは重要です．したがって，もし決定的プログラムについてうまく理由付けができれば，非決定的なプログラムについても，それぞれ分けて考えることによって理由付けることができます．これにはさらに多くの仕事を必要としますが，別の問題です．

すでに知っている非決定性が ALT と相互作用する法則を使ってみましょう．チャネル a か b のどちらかで非決定的に出力しているプロセスを考えてみます．これにチャネル b 入力プロセスを並列に入れて，分配の法則と代入の法則によって計算します．

```
PAR      =    OR       =    OR
  OR             PAR            x:= e
    b ! e          b ! e
    c ! e          b ? x
  b ? x          PAR              STOP
                 c ! e
                 b ? x
```

この展開はプログラム中の固有の危険性を示します．もしプログラムが幸運にもうまく走れば，代入の結果が得られますが，うまくいかなければ deadlock となります．だから優秀なエンジニアでもうまくいくとは限りません．彼はその場合，プログラムの設計のはじめから正確に動作することを確かめたいだろうと思いますが…．そのために occam の法則があり，またプログラマにとっては大変それが役に立つのであります．

もしあなたのプロセスがチャネル b または c に出力するかどうか本当に知らなければ，あなたの期待はどちらかの可能性を待っているだけです．明確に設計された occam のコンストラクションで，これを許しているのは ALT であります．そして，プロセスが二つのチャネル b または c のどちらかで入力状態の場合，同時に待たせることができます．そして最後に，出力プロセスを介して終了する．このことは代数的形式（specification）によって表わされます．

```
PAR    =    OR    =    OR = x:= e
     OR               PAR            x:= e
       b ! e            b ! e
                        ALT
       c ! e            b ? x
                        c ? x
     ALT                PAR            x:= e
       b ? x            c ! e
       c ? x            ALT
                          b ? x
                          c ? x
```

deadlock だけでなく非決定性も回避されました．そして，プログラム全体の展開は皮肉なからませ（twist）から成り立っています．元来，非決定性に導入された ALT コンストラクションは非決定性を制御したり服従させるのに必要な道具となります．ALT は occam の最も独特で有益な特徴の一つであります．この代数の性質における小さな例を通してさらに深く理解され，またプログラムの中にもっと有効的に使われることを期待致します．

しかし，これは occam プログラミングのほんの一部であります．見てわかるように，おのおのの法則は他の法則とは独立して簡単に記述されています．この法則は他の言語の法則と結合することによって最も効

果的になります．悪いことには百以上の法則があり，すべての法則は算術の代数法則と同じ方法で数学的理論として証明されています．それらは Bill Roscoe によって集約されています．occam プログラミングの法則は二つの有限なプログラムが数学的に同等な意味かどうか，そして実行するときに，論理的に同じ効果なのか証明するのに非常に強力です．もちろん，タイミングを無視しなければなりまん．もしプロセスのタイミングを改良したいと思うなら，タイミングは正に数学が逆説的に無視しなくてはならないものです．

Bill Roscoe と Oxford における彼のチームは，Alvey プロジェクトを Inmos と共同で Transputer を異なった種類のコンフィギュレーションによって最適に実行したり，あるいは ASIC のように直接設計できるマシンに occam プログラムの変換システムを ML (Meta Language) の中で開発していました．これは，ハードウェアとソフトウェアの必要で実効的な組合せを得るためには特に有益でした．たとえば，並列プログラムを occam プログラムで書いたとしましょう．たぶんそれは，複数のトランスピュータ・ネットワーク上でうまく走っているでしょう．しかし，私たちはそれを特別のハードウェアに取り付けて 100 倍速くしたくなります．その場合，ハードウェアの部品はたぶん組合せ論理回路になったり，PLA，特別な機能ユニット，またはマイクロコードによる制御プログラムになるでしょう．しかし，プログラムの主な管理上の部分はハードウェアではなくて，むしろ汎用性のある Transputer によるソフトウェアで実行される部分が残ります．

設計の戦略が，徐々におのおのの occam プログラムからハードウェアの一部と同等の動作をシミュレーションしている等価な occam プログラムへと変換されています．簡単な例として，ブーリアン (Boolean) 表現は明らかに組合せ回路の記述です．そして自動的にハードウェア設計に変換することができます．PLA は入力にスタート，出力にエンドとなる occam の無限ループによって明確に記述できます．この場合，すべての変数は論理型でなくてはなりません．

```
WHILE TRUE
  SEQ
    in ? b, c, d …
    …
    out ! f, g, h
```

一連の変換の結果によって，元のアプリケーションプログラムはハードウェアに依存した言語を含むことになります．にもかかわらず，occam プログラムはまだ価値があるのです．それはまだ Transputer 上で走ることができるからです．走るとき実効速度はたとえ遅くとも，元のプログラムと同じ論理効果をもつことができます．しかし，一度プログラムに関連した部分がハードウェア化されると，それは何倍か速くなります．これが全体の目的です．そして，これは occam のプロセスの概念が十分に理論的なので，ハードウェア化への応用は成功します．

アプリケーションに特化したハードウェアの開発を妨げている主な要素は，開発の失敗に対する危険性です．さらに改良には多くの資金が必要となり，市場に参入するのが遅れます．occam の変換システムは，途中で自動設計または occam 言語を正しく検証した代数法則への確認の一致をとることにより，この危険性を取り除けるように設計されています．正確に組み立てられていると信じられているのは，個人的な回路だけではありません．これと同じことはインタフェース，特にエラーが潜んでおり，そのエラーが増大するかもしれないと思われる場所であるハードウェアとソフトウェアの間に関していえます．そしてこれは，occam と Transputer を設計した CSP の基本的な数学的根拠の理論的研究の最終的な償いです．

最初の大きなスケールでしかも実用的な occam の応用は，Transputer (T 800) の FPU 設計でした．FPU の算術的なアルゴリズムは Inmos の Roger Shepherd によって occam プログラムでコーディングされ，Transputer の配列接続で高速にテストされました．テスト結果は市販の FPU チップと比較され，結果は正しいと思っていたのが実は誤っていました．そこで Oxford の Geoff Barrett が，プログラムが正確であることを数学的証明を使って試みました．彼は最初に，IEEE 浮動小数点の標準を命題の計算 (predicate calculus) の表明 (assertion) として仕様を形式化しました (formalize)．次に彼は，正確にインプリメントする有名な方法として証明と正確さを共

に考慮しました．さらに，効率の良い浮動小数点のルーチンが自分の証明したコードと論理的に等価であることを確かめるために，最後に occam プログラミングの法則を適用しました．失敗したときにはエラーフラグを立ててプログラムの動作を確認しました．3か月かかって発見されるエラーを彼は3週間で検出することができました．これをきっかけとして Inmos は，勇敢にもこのときから証明に基づいた設計方法を信頼するようになりました．

次の仕事はもっと厄介なもので，FPU のハードウェアのマイクロコード制御の正確さを証明することでした．マイクロコードは occam プログラムから翻訳されます．この occam の翻訳システムは Inmos の David Shepherd によって考案され，証明されたソフトウェアルーチンと同じ効果をもつことが確立されました．この仕事は最初のシリコンが出るまでに完成し，設計がエラーフリーであることが確かめられました．そして事実，それ依頼エラーは見つかっていないようです．たぶんご存知でしょうが，このプロジェクトは，Inmos と Oxford 大学 Computing Laboratory との共同プロジェクトで，はじめての Queen's Award を受賞したものです．

いま Oxford において私たちは，ヨーロッパと共同で ESPRIT プロジェクトのスポンサーの下で組込み型リアルタイムシステムに重点をおく（ハードウェア，ソフトウェアを含む）完全に証明されたシステムへの基礎的な研究に従事しています．その中で必須なものは，高級言語のためのコンパイラの正確な証明です．私たちはオブジェクトマシンとして Transputer，またサブセットとして occam をプログラミング言語として選択しました．オブジェクトコードの意味はインタープリタによって定義され，これもまた occam で書かれています．したがって，元のソースコードと同じ論理効果をもつコンパイラによって生成されたオブジェクトの通訳を示すのに occam プログラムの法則を使うことができます．私たちは現在，このテクニックをもう少し大きなサブセットの occam（並列，タイミングを含む）へ拡張しようとしています．しかし私たちの責任分野はコンパイラの正確さにだけ求められており，いまのところの見通しは全体の半ばです．一つにはリアルタイムで必要とされる方法（危険性の分析を含む），そして信頼性のある occam プログラムのための仕様への変換を見ていくつもりです．最後に Cambridge 大学の Mike Gordon と協力し，その理論を拡張し，ハードウェア化を検討しています．私たちは，テキサス州 Austin にある Computational Logic Inc.の Don Good が同様なプロジェクトを行なっているのに元気づけられました．

これで Transputer と occam の開発に選択的でランダムに関与してきた話は終了します．過去25年間ハードウェアの進歩は著しく発展しました．私がコンピュータ（Elliott 803）を始めたとき，計算機の速度は人間より1万倍速いものであり，1台29,000ポンドで2000台売れました．Transputer は Elliott 803 より1万倍速くて，同じ値段なら1,000個の Transputer を購入できます．値段は1/10000となり，驚くまでもなくすでに200,000個の Transputer が出荷されています．

ソフトウェアの進歩も大きく変わりますが，あまり劇的ではありません．occam 言語では Algol 60 の伝統を受け継いで設計され，オペレーティングシステムは Transputer のマイクロコードとして組み込まれています．しかし，ソフトウェアの進歩の尺度を1万倍あるいは10万倍として見るのは不可能です．にもかかわらずプログラミング言語，オペレーティングシステム，さらにそれらを信頼して使用する方法の理論的な理解等の質的な進歩はあったと思われます．基本的な知識の進歩は，改善された設計空間と設計方法による論理的な結果の意識によってソフトウェアとハードウェアの設計を可能にさせました．

数学的なモデルと法則への明白な注目は（10年以上前に研究を始めたときは期待していなかったが），ハードウェアとソフトウェアの信頼できる設計へと直接のアプリケーションを見ることによって償われました．Transputer のアプリケーションプログラムを書いている人たちが全面的に数学的な恩恵を受けているとは思いません．多くの人々は，occam 言語の意地の悪い制限を知っているだろうと思います．しかし高度に信頼性の高いプログラムを書いたり，異なったコンフィギュレーション，あるいは異なった技術でプログラムを走らせるには，プログラミング言語の下にある高度な抽象的概念に気づくでしょう．これが私たちの希望であり，たとえ10年以上かかっても研究をし続けていきます．

私の過去における話が非常に多くの部分に費やされたことを心配されるかもしれません．将来について，私はこれと同等の時間を費やすつもりがありませんので多少安心されるだろうと思います．この不釣り合いな理由は簡単です．将来を予測することは簡単です．それは過去とまったく同じことになるだろうと思われます．次の25年で，さらに同じ値段で1万倍スピードが速くなるでありましょうし，大きなスケールの並列と進歩した製造技術が得られるであろうと思います．しかし，値段は劇的には下がらないだろうと思います．ギガFLOPのコンピュータは当然29ペンスでは買うことができないでしょう．しかしながら，お金に対する価値は劇的に増大し，コンピュータハードウェアに費やされる金額は増加し続けるでしょう．

ソフトウェア側についていえば，過去における類似は優れたUNIXオペレーティングシステムとCプログラミング言語の将来を予測することができます．Fortranが Algol 60に浸されたように，Pascalやoccamの開発や使用がC言語によって浸されるでしょう．私の思うにそれは一種の進歩であるが，たぶん間違った方向であろうと思っています．つまりビジネスが失敗するということではなく，あまりにも深く浸ることになるだろうということです．日曜日の朝2時にオペレーティングのハッカーが突如頭に閃いた最初のアイディアが，15年間にわたりヨーロッパの進んだ研究所の基礎的な研究成果よりも実用的に影響を与えているとは実に悲しい事実です．しかし，これは科学や工学の基礎研究に貢献している人々にとっての一般的な宿命です．基礎研究が理解され，感謝され，そして最後に工学に応用されるには何年もの歳月が必要とされます．しかし最後にそれが実用化されると，信頼性，性能，そして値段の効果が十分発揮されるものです．

これが希望をもってTransputerに安心できる理由です．そのすばらしい能力は，リアルタイムの制御における強制がスーパーコンピュータ並みの能力を要求される組込み型アプリケーションに見られます．このコンファレンスにおける参加者と発表者の努力に感謝します．私の希望は実現できたと思います．これは私のような理論家ではなく，経済性の限界を越えたインタラクティブなアプリケーション領域のコンピューティングの利点を広めようとされている，みなさまのおかげだと思います．さらなる利点は，アルゴリズムの

クリティカルな時間を速くする特別目的のハードウェアに目を向けたときに得られます．

いくつかの正確さの保存に関する変換の設計の理論的開発と研究が，最初のシリコンを正しく作り上げるのに役立つことを希望します．それは時間だけでなく，お金の節約もできます．次の25年間で数学的な技術が安全でかつクリティカルなアプリケーションの工学的な実践の標準規約の中で具体化することを（たとえ十分でない技術を最初に標準化しようとする重大な危険性が存在していても）期待します．

最後に，汎用大型マイクロプロセッサネットワークに対する成功の見込みについてお話ししたいと思います．なぜ魅力的かと申しますと，それは経済的だからです．コンピュータに高い信頼性を与えるだけでなく，ユーザは単に多くのプロセッサを追加するだけで性能を増すことができます．さらに良いことは，更新のときはプロセッサは安く買ったときよりも速くなっているので，たぶん半分の値段で倍の効果が出ます．値段がさらに下がれば，はじめの頃のいくつかは交換することができます．ユーザは大型コンピュータのように，定期的な交換とそれに伴う高い経費を必要とせずに，効率の良い供給ができます．

この期待できる開発に対するただ一つの障害は，通常のマイクロプロセッサ，大型計算機，スーパーコンピュータなどで用意されている数多くのアプリケーションソフトウェア，ユーティリティ，言語，オペレーティングシステムなどが数少ないということです．

汎用のソフトウェアを作るのに難しい点は，現在のTransputer のネットワークでは大きなスケールの汎用コンピュータを作るのがまだ難しいからです．そのような物はまだ設計されておりません．アーキテクチャとプログラミング言語 occam の両方は最適化され，小さなスケールの組込み型リアルタイム処理の応用として使われています．これらはリンクおよびネットワークのメインメモリ分散のハードウェアの詳細を独立にさせる必要があるので，ライブラリのデザインは大変難しいのです．

したがって重要な質問は私たち，Inmos そしてたぶんヨーロッパの IT (Information Technology) に対してでありましょう．Transputer のアーキテクチャは本当の汎用計算機となりうるのだろうか？ 私は，将来の汎用マルチプロセッサは数千のプロセッサと数千の独立したメモリモジュールから構成されると信じています．それらは数百個のプロセッサとメモリモジュールを同時にアクセスできる，洗練されたスイッチによって接続されるでしょう．同時にスイッチは，数百のデータ転送を一方から他方へと自律的に実行できるスイッチを通過することができ，遅延時間は大容量のキャシュメモリをおのおののプロセッサにもたせることと，マルチプログラミングによって小さくすることができます．

私はすべてのプロセッサが同じ物になるとは期待しません．事実 IBM, HP, Motorola, SUN などの異なったマシンが，すべてネットワークの概念をもつことは大変魅力的です．しかし，私はすべてのプロセッサが Transputer のプロセスの概念をサポートできるためには，Transputer アーキテクチャの方向に発展することをお薦めします．そして Transputer 自身，このような用途として最も効率の良いアーキテクチャではないかと思っています．CSP の概念は，超並列マシンのプログラミングにおいても最も効率の良い原理です．（たとえこれが最もポピュラーでなくても，あるいは使いずらいマシンであっても）以下にその理由を述べます．

（1） 任意のスケール（10〜100万）のプロセスの並列処理において良い制御ができます．

（2） オーバヘッドのない実現的な粒度が得られます．

（3） 共有されたプロセスのメモリを更新する際の occam の制限は，結合ルックアサイド，スヌープキャッシュ方式の過度のオーバヘッドを避けるのに必須です．

（4） 二つのプロセス間の通信による同期は最小の書込み時間と遅延時間があり，共有されたセマフォ (semaphore) 操作の効率を低下させます．

（5） Transputer の非常に速い割込み処理と，プロセスの切換えは，マルチプログラミングによってマスクされたメモリアクセスの遅延（いわゆる弛緩）を克服するのに必須なことです．

（6） 同期による遅延は ALT コンストラクションを使用することで減じることができ，プロセ

ッサの待ち時間がさらに速くなります．

私は，Transputer のプロセス概念は occam プログラミング言語の中で実現されているように，次の25年間は正しいと思います．そしてたぶん，理解するには苦痛であろうが，その努力に対して報酬を受けるだろうと思います．なぜなら，概念が十分に多様な技術(ハードウェア，ソフトウェア，マルチプログラミング，point-to-point による通信，ネットワークスイッチ，超並列コンピュータにおける自律的なデータ転送によるマルチプロセッシングなどを含む) の中に効率良くインプリメントされるのに抽象的であるからです．

謝　辞

特権を持って仕事をされていた多くのコンピュータ開拓者たちのすばらしい貢献にお礼を申し上げます．聴衆の皆さんそして読者の方々，どうも有難うございました．

参考文献

1) C.A.R. Hoare: Report on the Elliott Algol Transrator, *BCS Computer Journal*, 5(2), 127-9.
2) C.A.R. Hoare: Proof of Correctness of Data Representations, *Acta Informatica*, 1(4), 271-81.
3) C.A.R. Hoare: Monitors: An Operating System Structuring Concept, *CACM*, 17(10), 547-57.
4) C.A.R. Hoare: A Structured Paging System, *BCS Computer Journal*, 16(3), 209-15.
5) Jim Welsh and Michael McKeag: *Structured System Programming*, Prentice/Hall International, Hemel Hempstead, pp. 424, 1980.
6) C.A.R. Hoare: The Structure of an Operating System in Language Hierarchies and Interfaces, Springer-Verlag LNCS No. 85, Berlin, 338-50, 1980.
7) C.A.R. Hoare: Communicating Sequential Processes, *CACM*, 21(8), 666-77.
8) C.A.R. Hoare and J.R. Kennaway: A Theory of Non-Determism, Proceedings ICALP '80, Springer-Verlag LNCS No. 85, Berlin, 338-350, 1980.
9) S.D. Brookes, C.A.R. Hoare and A. W. Roscoe: A Theory of Communicating Sequential Processes, *Journal of ACM*, 31(3), 560-99.
10) A.W. Roscoe and C.A.R. Hoare: The Laws of Occam Programming, Programming Research Group Technical Monograph PRG 53, Oxford University, 1986.

(まつい かずと
エス・ジー・エス・トムソン・マイクロエレクトロニクス(株))