CSP-Prover: プロセス代数 CSP のための定理証明器

○ 磯部 祥尚† Markus Roggenbach††

我々は汎用定理証明器 Isabelle 上にプロセス代数 Csp の理論を実装し、Csp のための証明支援ツール Csp-Prover を開発している。Csp-Prover には、完備距離空間と完備半順序の理論が実装されており、そこで Banach の不動点定理と Tarski の不動点定理が証明されている。これらは Csp において再帰動作を解析するために使われる重要な定理である。現在の Csp-Prover では、プロセス (動作) の意味は安定失敗モデル $\mathcal{F}$ を基に定義されており、その再帰動作は主に Banach の不動点定理によって解析されている。さらに、Csp の規則も多数証明されており、その Csp の規則を基にプロセスの詳細化関係を半自動的に検証するコマンドも用意されている。すなわち、Csp-Prover は Csp の新しいモデルや規則の証明のような研究を支援するだけでなく、実際の並行システムの検証支援にも有効である。特に無限の状態をもつプロセスの詳細化関係を不動点帰納法等によって検証できる特長をもつ。本発表では Csp-Prover について説明すると共に、実際の電子支払システム EP2 と無限状態をもつ数学者の食事問題に Csp-Prover を適用してその有効性を実証する。

CSP-Prover: a Theorem Prover for a Process Algebra CSP

○ Yoshinao Isobe† and Markus Roggenbach††

We are developing a proof-assistant tool called Csp-Prover for the process algebra Csp by embedding the theory of Csp into the generic theorem prover Isabelle. In Csp-Prover, both of the theory of complete metric spaces and the theory of complete partial orders are contained, and Banach's fixed-point theorem and Tarski's fixed-point theorem are proven. These theorems can be used for analysis of recursive processes. In the current Csp-Prover, the semantics of processes is defined based on the stable failures model $\mathcal{F}$, and recursive processes are analyzed mainly by Banach's fixed-point theorem. Then, many Csp-laws are proven, and also semi-automatic tactics are given based on the Csp-laws for proving a refinement relation between processes. Therefore, Csp-Prover is available not only for academic researches such as proofs of new models and laws, but also for verification of practical systems. Especially, infinite state systems can be verified by fixed-point inductions. In this presentation, we introduce Csp-Prover and demonstrate the availability by applying it to verification on a practical electronic payment system called EP2 and a classical example called the dining mathematicians of infinite state systems.

1. はじめに

プロセス代数[2] は並行システムを記述し、その動作の正しさを検証するための形式的な枠組である。Csp (Communicating Sequential Processes)[9),17] は代表的なプロセス代数の一つであり、鉄道システム[3] やセキュリティプロトコル[18] の検証等、様々な分野で利用されている。

モデル検査器 FDR[11] は Csp のための代表的な検証ツールであり、仕様の詳細化関係の検証をはじめ、デッドロックやライブロックの検出等も可能である。ただし、一般に FDR はシステムの全ての状態を探索するため、その状態数が無限である場合や、無限でなくても非常に大きい場合は現実的には検証できないという問題がある。また、変数や関数は全て明確に定義されていなければならない制約がある。このような問題を補うために、プロセス代数の検証に定理証明器を使う方法が提案されている[6),19)~21]。

我々は汎用定理証明器 Isabelle[12] 上に Csp の理論を実装し、Csp のための証明支援ツール Csp-Prover を開発している。現在の Csp-Prover は、セーフティ特性やライブネス特性の検証に適した安定失敗モデル $\mathcal{F}$ をもとに実装されているが、Csp-Prover の構成は汎用部分とモデル依存部分に分かれており、汎用部分は他のモデルに対しても再利用できる。その汎用部分

† 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology
†† ウェールズ大学スウォンジー校
University of Wales Swansea

には無限状態システムを帰納的に解析するために有効な完備距離空間と完備半順序の理論が含まれている。また、Isabelle では抽象的なデータ型を扱うことができるため、システム開発の初期の仕様に見られるような曖昧な変数や関数を記述できる特長がある。

本稿では、まず 2 節と 3 節で定理証明器 Isabelle とプロセス代数 Csp の概要を述べる。次に、4 節で Csp-Prover の構成について説明し、5 節で Csp-Prover の適用例を紹介する。最後に Csp-Prover と関連研究を比較する。

2. 汎用定理証明器 Isabelle

Isabelle[14] は対話型の定理証明器である。利用者は証明する定理を**ゴール**として入力し、**証明コマンド**を与える。Isabelle は与えられた証明コマンドに従って**書き換え規則**をゴールに適用し、その結果を新しいゴールとして表示する。利用者はその結果をもとに、次の証明コマンドを与える。最終的にゴールが真値 `True` に書き換えられたときに証明は完了する。証明が完了した定理は新しい規則として Isabelle に蓄えられ、今後の他の定理の証明に利用可能となる。

Isabelle の能力を拡張するために、型、関数、述語等を追加定義することができる。例えば、次のようにキーワード **typedef** によって、新しい型を既存の型の部分集合 (非空) として定義できる：

```
typedef SubType = {x::SuperType.  P(x)}
```

ここで、`P` は既存の型 `SuperType` 上の述語、`SubType` はその部分集合によって新しく定義される型である。また、キーワード **datatype** は型構成子を用いて再帰的な型を定義するために使われる。例えば、リスト型は次のように定義できる：

```
datatype
  'a list = Nil                  ("[]")
          | Cons 'a "'a list"    ("_ # _")
```

ここで、`Nil` と `Cons` が型構成子であり、`("_ # _")` は `(Cons a t)` を `(a # t)` のように表記することを可能にする。また、`'a` はリストの要素の型変数である。Isabelle では型変数をこの例のように前に `(')` を付けて区別する。

さらに、ある条件を満たす**型のクラス**を次のように定義することができる：

```
axclass SubClass < SuperClass
  name1: (a condition)
  name2: (a condition)
  ...
```

ここで、新しく定義される `SubClass` は、既存のクラス `SuperClass` に含まれ、かつ $\mathtt{name}_{1,2,\cdots}$ の名前をもつ条件を満たす型のクラスであり、`SuperClass` の全ての特性が `SubClass` によって継承される。また、ある型 `T` があるクラス `C` に属することはキーワード **instance** によって宣言できる。ただし、その宣言には、その型 `T` がそのクラス `C` と全ての親クラスの条件を満たすことを証明する必要がある。

定理や補題を、その証明に必要な定義や証明スクリプトとともに**論理ファイル**に記述することができる。Isabelle は論理ファイルを読み込み、データベースに蓄えることによって、他の論理ファイルからの参照を可能にしている。そのような論理ファイルは一般に次のフォーマットをもつ：

```
theory T = B1 + ··· + Bn:
  (宣言), (定義), (証明)
end
```

ここで、$\mathtt{B}_1, \cdots, \mathtt{B}_n$ は論理 `T` の**親論理**であり、親論理で定義された型や証明された定理等は全て子論理でも有効である。このように、論理ファイルは一般に階層構造をもつ。

3. プロセス代数 CSP

本節では、先ず Csp-Prover で使われている Csp の構文と意味を紹介し、次に再帰動作を解析する方法について説明する。この説明は主に文献 17) に基づいている。

3.1 構　　文

図 1 に Csp-Prover で使われている Csp プロセスの構文を示す。ここで、`'a` はプロセス間で渡される値や通信チャンネルの名前から構成される**通信要素**の型、`'n` は再帰動作を定義するための**プロセス名**の型である。図中、`'v` はチャンネル `c` によって渡される値の型であり、この型 `'v` はチャンネル毎に決めることができる。

Csp には外部選択 `[+]` と内部選択 `|~|` の二種類の選択演算子が用意されており、外部選択は他のプロセスやユーザによる選択、内部選択は外部からは非決定的にされる選択を意味している。また、これらの選択演算子を各々論理積と論理和のように使い、要求を記述することもできる。例えば、次の二つの仕様 `Spc1` と `Spc2` について、

```
Spc1 = (a -> SKIP) [+] (b -> SKIP)
Spc2 = (a -> SKIP) |~| (b -> SKIP)
```

`Spc1` は `a` かつ `b` を実行できる (実行後は成功終了) ことを要求し、`Spc2` は `a` または `b` を実行できることを

```
P ::=  SKIP                   %% 成功終了
     | STOP                   %% デッドロック
     | a -> P                 %% アクションプレフィクス                       (a::'a)
     | c ! v -> P             %% チャンネル c へ値 v の送信 (*)              (c::('v⇒'a), v::'v)
     | c ? x : X -> P(x)      %% チャンネル c から値 x∈X の受信 (*)         (c::('v⇒'a), x::'v, X::('v set))
     | c !! x : X -> P(x)     %% チャンネル c へ非決定的な値 x∈X の送信 (*)  (c::('v⇒'a), x::'v, X::('v set))
     | c !! x -> P(x)         %% チャンネル c へ非決定的な値 x の送信 (*)    (c::('v⇒'a), x::'v)
     | ? x : X -> P(x)        %% 外部プレフィクス選択                         (x::'a, X::('a set))
     | ! x : X -> P(x)        %% 内部プレフィクス選択 (*)                     (x::'a, X::('a set))
     | P [+] P                %% 外部選択
     | P |~| P                %% 内部選択
     | ! x : X ..  P(x)       %% 値 x∈X の非決定的選択                       (x::'a, X::('a set))
     | IF b THEN P ELSE P     %% 条件分岐                                     (b::bool)
     | P |[X]| P              %% 汎用並行合成                                 (X::('a set))
     | P ||| P                %% 独立並行合成 (*)
     | P || P                 %% 同期並行合成 (*)
     | P -- X                 %% 隠蔽                                         (X::('a set))
     | P [[r]]                %% 名前変更                                     (r::(('a * 'a) set))
     | P ;; P                 %% 逐次合成
     | P [> P                 %% タイムアウト (*)
     | <C>                    %% プロセス名                                   (C::'n)
```

図 1　Csp-Prover で使われている Csp の構文

要求している。このような外部選択と内部選択の概念はプレフィクス (->) にも反映され、図 1 に示すように 7 種類のプレフィクスが用意されている。

以下、他の演算子について簡単に説明する。汎用並行合成 (P |[X]| Q) は P と Q を独立に、ただし X に含まれる通信要素では同期するように並行合成する。独立並行合成と同期並行合成はその特殊な場合 (各々、X = {} と X = UNIV) である。隠蔽 (P -- X) は X に含まれる通信要素を外部から隠す。これによって外部選択が内部選択になる場合もある。名前変更 (P [[r]]) は r に従って通信要素名を変更する。例えば、(a,c),(b,c)∈r ならば P の通信要素 a と b を c に変更する。変更後の c による動作は元の a または b の動作に従う。a または b は非決定的に選択される。逐次合成 (P ;; Q) は P が SKIP で成功終了した後、Q のように振る舞う。タイムアウト (P [> Q) は ((P |~| STOP) [+] Q) の略記であり、P は非決定的に実行できなくなることを表している。タイムアウトの例のように、(*) によってマークされている演算子は他の演算子の略記法によって定義されている。

再帰プロセスは (LET df IN P) の形をもち、プロセス P に含まれるプロセス名 C の動作は関数 df から定義される等式の不動点によって与えられる。このような関数は、Isabelle のキーワード primrec を使って簡単に定義できる。例えば、増加する自然数 n をチャンネル c に繰り返し送信する再帰プロセス Inc は次のように定義できる：

```
primrec
  df   (Loop n) = c ! n -> <Loop (n+1)>
defs "Inc_def:
  Inc == LET df IN <Loop 0>"
```

このとき<Loop n>の意味は、全ての自然数 n について、プロセス<Loop n>と (c ! n -> <Loop (n+1)>) の意味が等しくなるように与えられる。このように変域が無限のパラメータをもつ再帰プロセスの意味は、無限に多くの等式の解によって与えられる。

3.2　意　　味

Csp の表示的意味論には、検証する特性に応じて様々なモデルが用意されている[17]。代表的な Csp のモデルとしては、セーフティ特性 (要求以上の動作をしない) の検証に適した**トレースモデル** $\mathcal{T}$、セーフティ特性に加えてライブネス特性 (要求の動作をする) の検証とデッドロック検出に適した**安定失敗モデル** $\mathcal{F}$、さらにライブロックの検出も可能な**失敗発散モデル** $\mathcal{N}$ がある。

現在の Csp-Prover は安定失敗モデル $\mathcal{F}$ をもとに実装されている。全ての通信要素の集合を A とすると、モデル $\mathcal{F}$ のドメインは、ある条件 **HC** (Healthiness Conditions) を満たすトレースの集合 $T \subseteq A^{*\checkmark}$ と**失敗**の集合 $F \subseteq A^{*\checkmark} \times \mathbb{P}(A^{\checkmark})$ の組 (T, F) の集合である☆。ここで "失敗" とはトレース t と通信要素の集合 X の組 (t, X) であり、これはトレース t を実行後に X に含まれる通信要素の実行に失敗する (実行できない) ことを表している。条件 HC は、ドメインを各モデルに適するように構築するための条件であり、モデル $\mathcal{F}$ のための条件 HC は T1, T2, T3, F2, F3, F4 である[17]。例えば、条件 T3 は (SKIP による) 成功終了後は全ての通信要素の実行に失敗することを要求している。

☆ $A^{\checkmark}$ は通信要素の集合に**成功終了イベント** $\checkmark$ を加えた**イベント**の集合 $A \cup \{\checkmark\}$ であり、$A^{*\checkmark}$ は全ての**トレース**の集合 $A^* \cup \{s \frown \langle\checkmark\rangle \mid s \in A^*\}$ である。

$$\begin{aligned}
traces(\texttt{STOP}) &= \{\langle\rangle\} \\
failures(\texttt{STOP}) &= \{(\langle\rangle, X) \mid X \subseteq A^{\checkmark}\} \\[1ex]
traces(\texttt{SKIP}) &= \{\langle\rangle, \langle\checkmark\rangle\} \\
failures(\texttt{SKIP}) &= \{(\langle\rangle, X) \mid X \subseteq A\} \cup \{(\langle\checkmark\rangle, X) \mid X \subseteq A^{\checkmark}\} \\[1ex]
traces(\texttt{a -> P}) &= \{\langle\rangle\} \cup \{\langle \texttt{a}\rangle \frown s \mid s \in traces(\texttt{P})\} \\
failures(\texttt{a -> P}) &= \{(\langle\rangle, X) \mid \texttt{a} \notin X\} \cup \{(\langle \texttt{a}\rangle \frown s, X) \mid (s, X) \in failures(\texttt{P})\} \\[1ex]
traces(\texttt{? x:A -> P}) &= \{\langle\rangle\} \cup \{\langle \texttt{a}\rangle \frown s \mid a \in \texttt{A} \wedge s \in traces(\texttt{P[a/x]})\} \\
failures(\texttt{? x:A -> P}) &= \{(\langle\rangle, X) \mid X \cap \texttt{A} = \{\}\} \cup \{(\langle \texttt{a}\rangle \frown s, X) \mid a \in \texttt{A} \wedge (s, X) \in failures(\texttt{P[a/x]})\}
\end{aligned}$$

図 2　安定失敗モデル $\mathcal{F}$ のための意味定義の例

プロセスの意味は各プロセスを各モデルのドメインの要素に写像することによって与えられる。モデル $\mathcal{F}$ についての写像 (*traces*, *failures*) の定義の一部を図 2 に示す。現在の Csp-Prover にはトレースモデルと安定失敗モデルの意味が定義されており、これらは文献 17) に与えられている定義と同じである。

3.3　CSP における再帰動作の解析

3.1 小節で述べたように、再帰的な動作は等式の解として表現される。例えば、イベント a を繰り返し実行できるか、または b によって成功終了できるプロセス<P>と常に a を繰り返し実行できるプロセス<Q>は次のように定義できる：

```
primrec df1   (P) = a -> <P> |~| b -> SKIP
primrec df2   (Q) = a -> <Q>
```

このとき、例えばプロセス名 P の意味は、プロセス<P>とプロセス (df1 P) の意味が等しくなるように与えられる。そこで考慮しなければならない問題は、(1) そのような等式の解が (例えばモデル $\mathcal{F}$ において) 存在するのか、(2) そのような解は唯一に定まるのか、(3) その解についての特性 (例えば、<Q>は<P>の詳細化であること) をどのように解析するのか、ということである。これらの問題を扱うために、Csp では二つの方法が利用されている、すなわち、**完備距離空間 (cms)** と**完備半順序 (cpo)** による不動点の解析方法である。

上記の問題 (1) に対しては、cms では **Banach の不動点定理**、cpo では **Tarski の不動点定理**を適用できる。Banach の不動点定理を適用するためには、与えられた Csp モデルのドメインが cms であることを証明し、各演算子が縮小写像 (contraction map) になることを証明する。同様に、Tarski の不動点定理を適用するためには、そのドメインが cpo であることを証明し、各演算子が連続になることを証明する。問題 (2) については、Banach の不動点定理は唯一解を保証している。一方、Tarski の不動点定理は唯一解を保証していないため、最小不動点が解として選ばれている。最後に問題 (3) については、不動点上の特性を解析するために、不動点帰納法と呼ばれる方法が cms と cpo に各々用意されている。

上記の不動点解析手続きは単一の等式について述べてきたが、積空間 (product space) を用いることによって、無限に多くの等式にも拡張できる。そのような無限に多くの等式は 3.1 小節で紹介した無限状態をもつ再帰プロセス Inc を解析するために必要である。

現在の Csp-Prover で採用されているモデル $\mathcal{F}$ の意味ドメインは通信要素数が無限の場合でも完備距離空間 (cms) かつ完備半順序 (cpo) となるため☆、再帰プロセスの解析に cms と cpo の両方の方法を適用できる。

4.　CSP-Prover の概要

Csp-Prover は汎用定理証明器 Isabelle[14] の理論 HOL-Complex☆☆上に図 3 に示す階層的な Csp の理論ファイルを追加して構築されている。Csp-Prover は Csp モデルに依存しない**汎用部分** $\texttt{Th}_{1,\cdots,6}$ と、各モデルのための**モデル依存部分** $\texttt{Th}_{7,8,9}$ に分割できる。すなわち、現在の Csp-Prover は安定失敗モデル $\mathcal{F}$ (必然的にトレースモデル $\mathcal{T}$ を含む) が実装されているが、他のモデルの実装に対しても汎用部分を再利用できる。

汎用部分は、cms に基づく Banach の不動点定理と (距離の概念による) 不動点帰納法、cpo に基づく Tarski の不動点定理と (標準の) 不動点帰納法を提供している。さらに、無限状態システム解析のために、cms と cpo を無限積空間に拡張する方法も与えられている。すなわち、Csp-Prover に新しく Csp モデルを追加する場合は、そのモデルのドメインが cms または cpo であることと、各々の条件 (演算子の縮小性や連続性) を満たすことを証明することによって、不動点

☆ $\mathcal{N}$ の意味ドメインは通信要素数が無限の場合は詳細化関係について cpo とはならない。ただし、cms にはなる。

☆☆ HOL-Complex は理論 HOL[12] に実数と複素数の理論を加えて拡張した理論である。

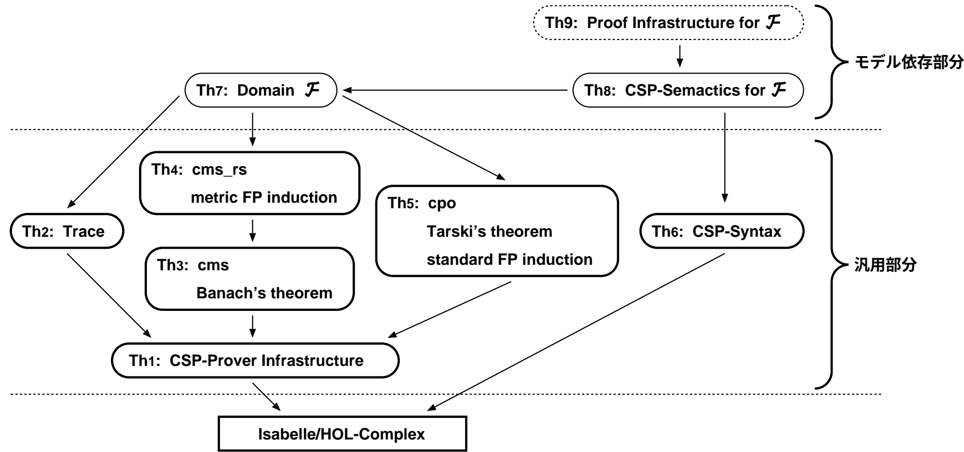

図 3　安定失敗モデル $\mathcal{F}$ を実装した Csp-Prover の理論マップ

定理、不動点帰納法、無限積空間への拡張等を新しいモデルでも利用可能になる。また、汎用部分にはドメインが cms または cpo であることを証明するために有用な補題が多数用意されている。特に、Csp のドメインが cms であることを証明するために、制限空間 (restriction spaces) の有効性が示されており[17]、その理論も汎用部分に含まれている。

汎用部分のもう一つの役割は、Csp の構文を再帰型として与えることである。すなわち、Csp-Prover に新しいモデルを導入したときは、この再帰型からそのモデルのドメインへの写像を定義することによってプロセスの意味を与える。このような定義の方法はディープエンコーディングと呼ばれ、プロセスの構造に関する帰納法を利用できる特長がある。

4.1 汎用部分

汎用部分は、トレース理論 ($\mathtt{Th}_2$)、cms と cpo についての不動点理論 ($\mathtt{Th}_{3,4,5}$)、Csp の構文定義 ($\mathtt{Th}_6$)、極限や上限についての補題等 ($\mathtt{Th}_1$) から構成されている。トレース型はリスト型の部分集合☆として与えられ、その性質は似ている。構文定義については 4.3 小節で意味定義とともに説明する。本小節では、cms と制限空間の実装について述べる。cpo の実装については文献 20), 21) に説明されている。

論理ファイル $\mathtt{Th}_3$ では、先ず距離空間が、最小性、対称性、三角不等式を満たす空間の型クラス ms として定義される。次に、型クラス ms のサブクラスとして、完備性の条件 complete_ms を追加して完備距離空間の型クラス cms が定義される：

```
axclass cms < ms
  complete_ms: "∀xs. cauchy xs
                 ⟶(∃x. xs convergeTo x)"
```

完備性の条件は、任意の Cauchy 列はある点に収束することを要求している。そして、Banach の不動点定理が証明される：

```
theorem Banach_thm:
 "contraction (f::('a::cms⇒'a))
  ⟹ f hasUFP ∧
     (λn. (f^n) x0) convergeTo (UFP f)"
```

すなわち、cms 上の任意の縮小写像 f は唯一の不動点をもち、任意の値 x0 に f を繰り返し適用して得られる列はその不動点に収束する。

Csp においては、各モデルのドメインが距離空間であることを証明するために、**制限空間**を利用する方法が知られており[17]、その方法が論理ファイル $\mathtt{Th}_4$ に与えられている。例えば、ある型が制限空間の型クラス rs のインスタンスであるならば、それは自動的に型クラス ms_rs のインスタンスとなることが証明されている。ここで、ms_rs は ms と rs の両方を継承する型クラスである。この型クラス ms_rs に関する重要な結果として、ms_rs の完備性が型構成子* (binary product) と fun (function space) によって保存されることがある。例えば、型 T が cms_rs のインスタンスであるとき、任意の型 I についてその関数型 I ⇒ T も cms_rs のインスタンスとなる。この定理は $\mathtt{Th}_4$ に次のように与えられている：

```
instance fun :: (type,cms_rs) cms_rs
```

この関数型 I ⇒ T は、単一空間 T をインデックス集合 I によって拡張した無限積空間とみなせる。3.3 小節

☆ Csp のイベント型は通信 (Ev a) と成功終了 ✓ から構成され、トレース型はイベントを要素にもつリスト型 (ただし、✓ は最後以外表れるない) である。

```
typedef 'a domT = "{T::('a trace set).  HC_T1(T)}"
types 'a failure = "'a trace * 'a event set"
typedef 'a domF = "{F::('a failure set).  HC_F2(F)}"
types 'a domTF = "'a domT * 'a domF"
typedef 'a domSF = "{SF::('a domTF). HC_T2(SF) ∧ HC_T3(SF) ∧ HC_F3(SF) ∧ HC_F4(SF)}"
```

図 4　安定失敗モデル $\mathcal{F}$ のドメイン domSF の定義

で述べたように、無限積空間は無限状態システムを解析するために必要である。安定失敗モデル $\mathcal{F}$ を例にとれば、任意のインデックス集合 I☆について、$\mathcal{F}^{\mathtt{I}}$ が cms であることが要求される。上記の定理は、$\mathcal{F}^{\mathtt{I}}$ が cms であることを証明する代わりに、$\mathcal{F}$ が cms_rs であることを証明すれば十分であることを示している。

最後に、cms_rs 上の述語 R について、関数 f の不動点への適用結果 (R (UFP f)) を帰納的に証明するための不動点帰納法が証明されている☆☆：

```
theorem cms_fixpoint_induction:
  "[| (R::'a::cms_rs⇒bool) x ;
      continuous_rs R ;
      constructive_rs f ;
      ∀x. R x → R (f x) |]
     ⟹ f hasUFP ∧ R (UFP f)"
```

4.2　モデル依存部分

モデル依存部分は、各モデルのドメイン理論 (図 3 の Th_7)、意味定義 (Th_8)、Csp の証明基盤 (Th_9) から構成されている。証明基盤はステップ規則 (展開規則)、分配規則、不動点帰納法等の多くの Csp 規則を提供している。さらにそれらの規則をもとに任意のプロセスをある標準形 **HNF**(Head Normal Form) に変換するための証明コマンド csp_hnf_tac も用意されている。これらの規則や証明コマンドの正しさは Isabelle によって証明されており、このことは Csp-Prover による検証結果が Csp の意味定義に関して健全であることを保証している。

現在、Csp-Prover には安定失敗モデル $\mathcal{F}$ とトレースモデル $\mathcal{T}$ のドメインが、各々型'a domSF と'a domT として定義されている。ここで、'a は通信要素の型である。また、図 4 にみられるように'a domT は'a domSF を定義するために再利用されている。図中、HC_T1, ···, HC_F4 は 3.2 小節で述べたモデル $\mathcal{F}$ のための条件 HC を実装した述語である。

Banach の定理とその不動点帰納法をモデル $\mathcal{F}$ に適用するために、'a domSF の無限積空間 ('i,'a) domSF_prod が cms_rs のインスタンスであることを証明する必要がある。ここで、型 ('i,'a) domSF_prod は ('i ⇒ 'a domSF) の略記であり、'i は積空間のインデックスの型である。この証明は次の手順で進められる：(1) domT と domF は cms_rs のインスタンスである、(2) domTF も cms_rs のインスタンスである、すなわち domSF (⊂ domTF) の各 Cauchy 列の極限が存在する、(3) その極限が domSF に含まれる、すなわち domSF も cms_rs のインスタンスである、(4) domSF_prod も cms_rs のインスタンスである。ここで、4.1 小節で述べたように cms_rs は型構成子*と fun によって保存されるので、上記の (2) と (4) を改めて証明する必要はない。この例は、新しいモデルが Csp-Prover に追加されたとき、制限空間がどのように証明の手間を削減できるかを示している。

4.3　ディープエンコーディング

Csp の構文は図 5 にみられるように Isabelle のキーワード **datatype** によって再帰的な型 ('n,'a) proc として与えられる。ここで、'n と'a は各々プロセス名と通信要素の型である。これは、Isabelle の証明コマンド induct_tac によってプロセスの構造に関する帰納法が簡単に適用できることを暗示している。また、この型 ('n,'a) proc は図 1 とほぼ同様の構文を与えるが、外部プレフィクス選択 (? x : X -> P) のように束縛変数 x をもつ演算子は、その変数を引数にもつ関数 Pf を用いて (? : X -> Pf) のように定義されている。これは **datatype** による型定義では束縛変数をもつ演算子 (型構成子) を直接定義できないためであるが、そのような演算子もシンタックスの変換を行う Isabelle のキーワード **syntax** と **translations** によって次のように簡単に定義できる。

```
syntax "@Ext_pre_choice" ::
  "pttrn ⇒ 'a set ⇒ ('n,'a) proc
     ⇒ ('n,'a) proc" ("? _ : _ -> _")
translations
  "? x : X -> P == ? : X -> (λ x. P)"
```

再帰プロセスは (LET:fp df IN P) の形をもち、その型は ('n,'a) procRC である。ここで、df は各プロセス名にプロセスを割り当てる関数であり、('n,'a) procDF の型をもつ。また、fp は値として Ufp か Lfp をとる変数であり、プロセス名に意味を与えるために

☆ 直感的に I は (無限に多くの) プロセス名の集合である。
☆☆ 制限空間における continuity や constructiveness については文献 17) に詳述されている。

```
datatype ('n,'a) proc = STOP
                      | SKIP
                      | Act_prefix     "'a" "('n,'a) proc"              ("_ -> _")
                      | Ext_pre_choice "'a set" "'a ⇒ ('n,'a) proc"     ("? : _ -> _")
                      | ···
                      | Proc_name      "'n"                             ("<_>")

type ('n,'a) procDF = "'n ⇒ ('n,'a) proc"

datatype fp_type = Ufp | Lfp

datatype
  ('n,'a) procRC = Letin "fp_type" "('n,'a) procDF" "('n,'a) proc"   ("LET:_ _ IN _")
```

図 5 Csp-Prover の構文定義

```
consts
 evalT  ::  "'a proc ⇒ ('n,'a) domSF_prod ⇒ 'a domT"  ("⟦_⟧T")
 evalF  ::  "'a proc ⇒ ('n,'a) domSF_prod ⇒ 'a domF"  ("⟦_⟧F")
 evalSF ::  "'a proc ⇒ ('n,'a) domSF_prod ⇒ 'a domSF" ("⟦_⟧SF")

primrec
 "⟦STOP⟧T          = (λe. {[]t}t)"
 "⟦SKIP⟧T          = (λe. {[]t, [✓]t}t)"
 "⟦a -> P⟧T        = (λe. {t. t=[]t ∨ (∃s. t=[Ev a]t @t s ∧ s ∈t ⟦P⟧T e) }t)"
 "⟦? : X -> Pf⟧T   = (λe. {t. t=[]t ∨ (∃a s. t=[Ev a]t @t s ∧ s ∈t ⟦Pf a⟧T e ∧ a ∈ X) }t)"
                          ⋮
 "⟦<C>⟧T           = (λe. fstSF (e C))"            (* note: fstSF (T ,, F) = T *)

primrec
 "⟦STOP⟧F          = (λe. {f. ∃X. f=([]t, X) }f)"
 "⟦SKIP⟧F          = (λe. {f. (∃X. f=([]t, X) ∧ X ⊆ Evset) ∨ (∃X. f=([✓]t, X)) }f)"
 "⟦a -> P⟧F        = (λe. {f. (∃X. f=([]t, X) ∧ Ev a ∉ X) ∨
                              (∃s X. f=([Ev a]t @t s, X) ∧ (s, X) ∈f ⟦P⟧F e) }f)"
 "⟦? : X -> Pf⟧F   = (λe. {f. (∃Y. f=([]t, Y) ∧ Ev`X ∩ Y = {}) ∨
                              (∃a s Y. f=([Ev a]t @t s, Y) ∧ (s, Y) ∈f ⟦Pf a⟧F e) ∧ a ∈ X) }f)"
                          ⋮
 "⟦<C>⟧F           = (λe. sndSF (e C))"            (* note: sndSF (T ,, F) = F *)

defs evalSF_def :
 "⟦P⟧SF == (λe. (⟦P⟧T e ,, ⟦P⟧F e))"

consts
 evalDF  ::  "('n⇒('m,'a) proc)⇒('m,'a) domSF_prod ⇒('n,'a) domSF_prod"   ("⟦_⟧DF")
 evalRC  ::  "('n,'a) procRC⇒'a domSF"                                    ("⟦_⟧RC")

defs evalDF_def :
 "⟦df⟧DF == (λe. (λC. (⟦df C⟧SF e)))"

recdef evalRC "measure (λx. 0)"
 "⟦LET:Ufp df IN P⟧RC = ⟦P⟧SF (UFP ⟦df⟧DF)"          (* based on cms *)
 "⟦LET:Lfp df IN P⟧RC = ⟦P⟧SF (LFP ⟦df⟧DF)"          (* based on cpo *)
```

図 6 Csp-Prover の意味定義

df のどのような不動点が使われるかを指定する：すなわち、Ufp で唯一の不動点、Lfp で最小不動点が使われる。現在、Csp-Prover のデフォルトは唯一の不動点であり、(LET df IN P) を (LET:Ufp df IN P) の略記法として用意している。

Csp の意味は、図 6 に示す写像 ($[\![P]\!]_{SF}$ e) によってプロセスの記述 P をドメイン 'a domSF の要素に変換することによって定義される。ここで、e は P に含まれるプロセス名の評価関数であり、プロセス<C>の意味は (e C) によって与えられる。この写像 ($[\![P]\!]_{SF}$ e) は二つの写像の組 ($[\![P]\!]_T$ e ,, $[\![P]\!]_F$ e) によって定義されている。ここで、組 (T ,, F) は (T, F) がモデル $\mathcal{F}$ の全ての条件 HC を満たすことを要求している。

各写像 ($[\![P]\!]_T$ e) と ($[\![P]\!]_F$ e) は文献 17) に与えられている意味の定義 (semantic clause) と同じになるように定義されている。尚、図 6 中の添字 t と f (例えば、$[]_t$、$\in_f$) は、list や set 等の Isabelle で既に使われている型に対する演算子と、domT と domF のための演算子を区別するために付けられている。

再帰的な動作を定義するためのプロセス名の関数 df の意味 $[\![df]\!]_{DF}$ は、各プロセス名 C の意味が $[\![df\ C]\!]_{SF}$ になるように与えられる。最後に、再帰プロセスの意

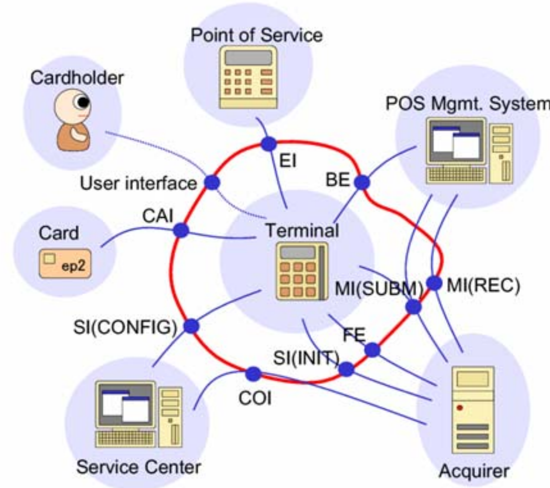

図 7 EP2 システムの構成

味 ⟦LET:fp df IN P⟧$_{\mathrm{RC}}$ が ⟦P⟧$_{\mathrm{SF}}$ によって与えられる。ここで、P のプロセス名の意味は、fp に応じて ⟦df⟧$_{\mathrm{DF}}$ の適切な不動点により与えられる。

5. CSP-Prover の適用例

本節で、並行システムの検証に Csp-Prover を適用してその有効性を示す。先ず、実際の電子支払システム EP2 の仕様書に対する詳細化関係の検証について述べる。次に数学者の食事問題と呼ばれる相互排他の例を扱う。

5.1 電子支払システム EP2 の検証

EP2 システム☆は電子支払システムの新しい工業標準の一つである。EP2 は図 7 に示すように、端末を中心に 7 つの自律的なプロセス (EP2 端末、カード保持者 (i.e. 顧客)、サービスポイント (e.g. キャッシュレジスター)、アクワイアラ、サービスセンター、カード) から構成されている。これらのプロセスは *XML* フォーマットのメッセージを互いに送受信している。

既に文献 7) において、EP2 システムのいくつかの部分が Csp-Casl ☆☆によって記述されている。EP2 のオリジナルの仕様書にしたがって、文献 7) で記述された形式的仕様はその抽象度によっていくつかのレベルに分類されている。例えば、アーキテクチャーレベル、アブストラクトコンポーネントレベル、コンクリートコンポーネントレベル等がある。そこで、このような異なるレベルの仕様書の詳細化関係を検証するためのツールが必要となっていた。

文献 7) で記述された Csp-Casl の仕様書を Csp-Prover で検証するために、我々はまず仕様書のデータ部分 (Casl コード) を適切な Isabelle の記述に書き換えて、Csp-Prover に適した入力フォーマットを得た。図 8 はアブストラクトコンポーネントレベルにおける EP2 端末の初期化手続きの重要な部分を抜き出した仕様である☆☆☆。Terminal は初期化リクエストを送信 (11 行目) した後、Acquirer から送られるデータを待つ。もしそのデータが Request の型を持つならば Terminal は Response 型の値を返信し (14 行目)、Exit の型を持つならば Terminal は初期化を成功終了する (15 行目)。それ以外の場合はデッドロック (STOP) する (15 行目)。他方、Acquirer は初期化リクエストを受信 (16 行目) した後、初期化を終了するか (18 行目)、またはリクエストとレスポンスを送受信するか (19 行目) を内部で決定する。解析される並行システムの仕様 AC は、チャンネル c で通信する Terminal と Acquirer の並行合成として定義される (22 行目)。

アブストラクトコンポーネントレベルの一つの特長として、データの型が明確に定義されていないことがある。Isabelle ではこのようなデータを 2〜5 行目の

☆ EP2 はクレジットカード、デビットカード、電子マネーのための基盤を定義するために複数の金融機関 (主にスイス) と企業によって創設された共同プロジェクトである (www.eftpos2000.ch)。

☆☆ Csp-Casl[16] は、データを扱う並行システムを形式的に解析するために、データ部分を形式的仕様記述言語 Casl で記述し、プロセス部分をプロセス代数 Csp で記述できるように Casl と Csp を統合した仕様記述言語である。

☆☆☆ 説明を分かりやすくするために本稿の仕様は簡略化されている。完全な仕様と証明は 10) から入手できる

```
 1  (* data part *)
 2  typedecl init_d      typedecl request_d
 3  typedecl response_d  typedecl exit_d
 4  datatype Data = Init init_d | Exit exit_d | Request request_d | Response response_d
 5  datatype Event = c Data
 6
 7  (* process part *)
 8  datatype ACName = Acquirer | AcConfM | Terminal | TerminalConfM
 9  consts ACDef :: "(ACName, Event) procDF"
10  primrec
11    "ACDef (Terminal) = c !! init:(range Init) -> <TerminalConfM>"
12    "ACDef (TerminalConfM) =
13        c ? x -> IF (x:range Request)
14                 THEN c !! response:(range Response) -> <TerminalConfM>
15                 ELSE IF (x:range Exit) THEN SKIP ELSE STOP"
16    "ACDef (Acquirer) = c ? x:(range Init) -> <AcConfM>"
17    "ACDef (AcConfM) =
18        c !! exit:(range Exit) -> SKIP |~|
19        c !! request:(range Request) -> c ? response:(range Response) -> <AcConfM>"
20
21  constdefs AC :: "(ACName, Event) procRC"
22    "AC == LET ACDef IN (<Acquirer> |[range c]| <Terminal>)"
```

図 8 アブストラクトコンポーネントレベルの EP2 仕様書 (一部)

```
 1  datatype AbsName = Abstract | Loop
 2  consts AbsDef :: "(AbsName, Event) procDF"
 3  primrec
 4    "AbsDef (Abstract) = c !! init:(range Init) -> <Loop>"
 5    "AbsDef (Loop) =
 6        c !! exit:(range Exit) -> SKIP |~|
 7        c !! request:(range Request) -> c !! response:(range Response) -> <Loop>"
 8
 9  constdefs Abs :: "(AbsName, Event) procRC"
10    "Abs == LET AbsDef IN <Abstract>"
```

図 9 仕様 `AC` の逐次的な仕様 `Abs`

ように宣言できる。このような型は後に無限の要素をもつ型に具体化 (詳細化) されることもできるため、`AC` は無限状態システムの可能性を含んでいる。例えば `init_d` が自然数型であれば、`Terminal` は無限の要素から非決定的に値を選び送信することになる (11 行目)。Csp-Prover では、抽象的な型を持つ仕様の特性が、それを具体化した任意の仕様にも継承されるため、抽象的な仕様を一つの仕様クラスのように扱うことができる。

Csp-Prover によって、我々は図 8 の仕様 `AC` が、図 9 の仕様 `Abs` に安定失敗等価 (安定失敗モデル $\mathcal{F}$ の意味での等価性) であることを証明した。ここで、`Abs` は逐次的な演算子のみで記述されているため、それがデッドロックをもたないことは記述から明らかである。すなわち、安定失敗モデル $\mathcal{F}$ による検証はデッドロックフリーを保存するので、この等価性はアブストラクトコンポーネントレベルの `Terminal` と `Acquirer` の通信はデッドロックを引き起こさないことを保証している。図 10 は Csp-Prover で等価性 `Abs =F AC` (14 行目) を証明するための完全なスクリプトである。まず、`Abs` のプロセス名を相当する `AC` のプロセスに変換する写像 `Abs_to_AC` を与える (3–5 行目) ☆。次に、再帰プロセス定義の各プロセス式内のプロセス名がプレフィクスの内側にある (ガードされている) ことと、隠蔽演算子を含まないことが確認される☆☆。これは簡単に自動的に証明される (8–11 行目)。これらの準備の後、`Abs =F AC` がゴールとして与えられる (14 行目)。`Abs` と `AC` の定義を展開した後 (15 行目)、上記の写像 `Abs_to_AC` と不動点帰納法によって、再帰プロセスを基底と帰納過程に展開する (16 行目)。基底は `simp_all` によって簡単に証明される。

帰納過程は `Abs` の各プロセス名について場合分けする必要があるが、これは `AbsName` に関する帰納法によって実行できる (17 行目)。最後に、Isabelle の自動証明コマンド `auto` と、標準形 HNF に変換する Csp-Prover の自動証明コマンド `csp_hnf_tac`、Csp 演算子を分割する規則 `csp_decompo` を繰り返し適用

☆ このような写像を完全に自動で導くことは困難である。ただし、このような写像を導くための支援ツールとして Csp-Prover を使うことはでき、実際にその有効性を確認している。

☆☆ これにより `ACDef` と `AbsDef` が縮小写像となり、その不動点解析に Banach の定理を適用することが可能となる。

```
 1  (* expected correspondence of process-names in Abs to AC *)
 2  consts Abs_to_AC :: "AbsName ⇒ (ACName, Event) proc"
 3  primrec
 4    "Abs_to_AC (Abstract) = (<Acquirer> |[range c]| <Terminal>)"
 5    "Abs_to_AC     (Loop) = (<AcConfM> |[range c]| <TerminalConfM>)"
 6
 7  (* guarded and no hiding operator *)
 8  lemma guard_nohide[simp]:
 9    "!! C. guard (ACDef C) & nohide (ACDef C)"
10    "!! C. guard (AbsDef C) & nohide (AbsDef C)"
11  by (induct_tac C, simp_all, induct_tac C, simp_all)
12
13  (* the main theorem *)
14  theorem ep2: "Abs =F AC"
15  apply (unfold Abs_def AC_def)
16  apply (rule csp_fp_induct_cms[of _ _ _ "Abs_to_AC"], simp_all)
17  apply (induct_tac C)
18  by (auto simp add: image_iff | tactic {* csp_hnf_tac 1 *} | rule csp_decompo)+
```

図 10 AC =F Abs のための完全な証明スクリプト.

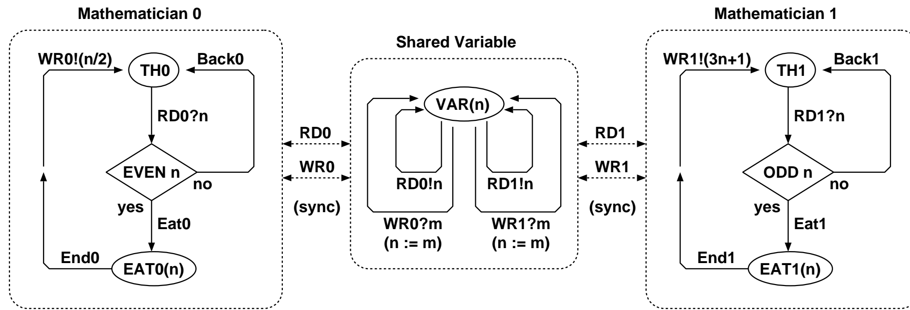

図 11 数学者の食事問題の状態遷移図

```
datatype Event = Eat0 | Back0 | End0 | RD0 int | WR0 int
               | Eat1 | Back1 | End1 | RD1 int | WR1 int | NUM int
syntax      "_CH0" :: "Event set" ("CH0")         "_CH1" :: "Event set" ("CH1")
translations "CH0" == "(range RD0) ∪ (range WR0)"  "CH1" == "(range RD1) ∪ (range WR1)"

datatype SysName = VAR int | TH0 | EAT0 int | TH1 | EAT1 int
consts SysDef :: "(SysName, Event) procDF"
primrec
   "SysDef       (TH0)    = RD0 ? n -> IF (EVEN n) THEN Eat0 -> <EAT0 n> ELSE Back0 -> <TH0>"
   "SysDef       (TH1)    = RD1 ? n -> IF (ODD n) THEN Eat1 -> <EAT1 n> ELSE Back1 -> <TH1>"
   "SysDef    (EAT0 n)    = End0 -> WR0 ! (n div 2) -> <TH0>
   "SysDef    (EAT1 n)    = End1 -> WR1 ! (3 * n + 1) -> <TH1>
   "SysDef     (VAR n)    =   WR0 ? n -> <VAR n> [+] WR1 ? n -> <VAR n>
                          [+] RD0 ! n -> <VAR n> [+] RD1 ! n -> <VAR n>"
constdefs Sys :: "int ⇒ (SysName, Event) procRC"
  "Sys == (λn.  LET SysDef IN (<TH0> |[CH0]| <VAR n> |[CH1]| <TH1>) -- (CH0 ∪ CH1))"
```

図 12 数学者の食事問題の Csp-Prover による記述

して証明は完了する (18 行目)。

5.2 数学者の食事問題

数学者の食事問題[5] は無限状態をもつ相互排他システムの例として知られている。この例は二人の数学者 (以下、各々数学者 0、数学者 1 と呼ぶ) と一つの共有変数から構成される並行システムとみなせる。各数学者は思考状態 (各々TH0、TH1) と食事状態 (各々EAT0、EAT1) をもち、共有変数には一つの整数 (n) が保存されている。ここで、二人の数学者は同時に食事することが無いように次のような取り決めをしている：数学者 0 は思考状態から共有変数の値 n を読み込み、n が偶数ならば食事を開始し、奇数ならば思考状態にもどる。そして食事が終了したときに共有変数に (n/2) を書き込む。他方、数学者 1 は奇数ならば食事を開始し、終了後は (3n+1) を書き込む。図 11 にこの二人の数学者と共有変数の状態遷移図を示す。図 12 はこの並行システムの Csp-Prover による記述例である。図中、Sys の引数 n は共有変数の初期値である。また、プロセス名 (EAT0 n), (EAT1 n), (VAR n) の引数 n は整数であり、このシステムは無限状態をもつ。

```
syntax "_OBS" :: "Event set" ("OBS")
translations "OBS" == "{Eat0, Back0, End0, Eat1, Back1, End1}"

datatype SpcName = TH0_TH1 | EAT0_TH1 | TH0_EAT1
consts SpcDef :: "(SpcName, Event) procDF"
primrec
   "SpcDef    (TH0_TH1) = ! x:OBS -> IF (x=Eat0) THEN <EAT0_TH1>
                             ELSE IF (x=Eat1) THEN <TH0_EAT1>
                                              ELSE <TH0_TH1>"
   "SpcDef   (EAT0_TH1) = ! x:(OBS - Eat1) -> IF (x=End0) THEN <TH0_TH1>
                                                      ELSE <EAT0_TH1>
   "SpcDef   (TH0_EAT1) = ! x:(OBS - Eat0) -> IF (x=End1) THEN <TH0_TH1>
                                                      ELSE <TH0_EAT1>
constdefs Spc :: "(SpcName, Event) procRC"
  "Spc == LET SpcDef IN <TH0_TH1>"
```

図 13　数学者の食事問題のための相互排他の仕様

この例で検証すべき問題は、二人の数学者が同時に食事しないこととデッドロック状態がないことを証明することである。より形式的には、数学者 0 が食事を始めて (Eat0) から終る (End0) まで数学者 1 が食事を始め (Eat1) ないこと (逆も同様) と、常になんらかのイベントが実行可能であることを証明することである。そこで、そのような要求を直接表現する逐次的な仕様 Spc を図 13 のように Csp-Prover で記述し、任意の初期値 n について、並行システム (Sys n) が逐次的な仕様 Spc の (モデル $\mathcal{F}$ の意味で) 詳細化であること (∀n. Spc <=F Sys n) を、Csp-Prover を用いて証明した。この仕様 Spc は、両方の数学者が思考状態の TH0_TH1、一方の数学者が食事状態で他方が思考状態の EAT0_TH1 と TH0_EAT1、の 3 つの状態をもち、相互排他とデッドロックフリーを明確に表している。その証明は 5.1 小節の図 10 よりも長くなるが、同様の手続きによって証明できる。すなわち、先ずゴールを不動点帰納法により展開し、プロセス式の標準形 HNF 変換とプレフィクスの分解を繰り返し実行する。その完全な記述は 10) から入手できる。

6. 関連研究

Isabelle[14]、HOL[8]、PVS[13] のような汎用定理証明器上に実装されたプロセス代数のための様々なツールが提案されている。

Csp-Prover に近い関連研究として、Tej/Wolff[20),21)] と Schneider/Dutertre[6),19)] による Csp 理論の実装がある。Tej/Wolff は Isabelle/HOL に cpo を基に Csp 理論を HOL-CSP として実装した。HOL-CSP では Csp の失敗発散モデル $\mathcal{N}$ が採用されているが、$\mathcal{N}$ のドメインは詳細化関係において cpo にならないため、より強いプロセス順序が使われている。そのため、5.2 小節で紹介した数学者の食事問題の詳細化検証には適用できない。また、構文がシャローエンコーディングされているため、プロセスの構造に関する帰納法は利用できない。Schneider/Dutertre はセキュリティプロトコルの検証を明確な目的として、Csp のトレースモデル $\mathcal{T}$ を PVS 上に実装した。ここでも再帰動作の解析には cpo が利用されている。この実装では、その明確な目的に必要ない演算子 (成功終了、隠蔽、名前変更、逐次合成) は省かれている。このため、5 節の例には適用できない。

表示的意味論ではなく、プロセス代数の公理的意味論を定理証明器に実装した研究も報告されている[1),4),15)]。ただし、このような公理をもとにした実装は Csp には適していない。

7. おわりに

我々は、定理証明器 Isabelle 上に Csp の理論を実装した Csp-Prover について述べてきた。関連研究と比較した Csp-Prover の主な利点はその汎用性にある。現在の Csp-Prover には安定失敗モデルをもとに基本的な演算子が全て用意されており、様々な並行システムを記述し、解析するための能力を提供している。それを実証するために、実際の電子支払システム EP2 や無限状態をもつ数学者の食事問題に Csp-Prover を適用してその有効性を示した。また、Csp-Prover の汎用部分には完備距離空間 (cms) と完備半順序 (cpo) を始めとする Csp 研究に必要な理論が含まれており、並行システムの検証支援ツールとしてだけでなく、Csp の新しいモデルや規則を証明するための研究支援ツールとしても利用できる。

今後は、失敗安定モデル $\mathcal{N}$ 等、他のモデルを Csp-Prover に実装してその能力を拡張するとともに、Csp のモデル検査器 FDR と Csp-Prover の統合を予定している。さらに、実際のシステム (EP2 や鉄道制御システム等) への Csp-Prover の適用を続け、より使いやすい証明コマンドの追加・修正を検討する。

謝辞 本研究を支援して頂いている Faron G. Moller 教授、大蒔和仁博士に感謝します。本研究を進めるうえで貴重な御意見を頂いている Erwin R. Catesbeiana (jr.) 氏、Christoph Lüth 博士、Ranko Lazic 博士、Jan Peleska 教授、Bill Roscoe 教授、Holger Schlingloff 教授に感謝します。また、本研究のために Royal Society (UK) より助成金を受けたことを記して感謝します。

参考文献

1) T. Basten and J. Hooman. Process algebra in PVS. In W. Cleaveland, editor, *TACAS'99*, LNCS 1579, pages 270–284. Springer, 1999.
2) J.Bergstra, A.Ponse, and S.Smolka. *Handbook of Process Algebra*. Elsevier, 2001.
3) B. Buth and M. Schrönen. Model-checking the architectural design of a fail-safe communication system for railway interlocking systems. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99*, LNCS 1709. Springer, 1999.
4) A. Camilleri. Combining interaction and automation in process algebra verification. In G. Goos and J. Hartmanis, editors, *TAPSOFT 1991*, LNCS 494, pages 283–295. Springer, 1991.
5) E. M. Clarke and H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2001.
6) B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In E. L. Gunter and A. Felty, editors, *TPHOL 1997*, LNCS 1275, pages 121–136. Springer, 1997.
7) A. Gimblett, M. Roggenbach, and H. Schlingloff. Towards a formal specification of an electronic payment system in Csp-Casl. In J.L. Fiadeiro, P.Mosses, and F.Orejas, editors, *WADT 2004*, LNCS. Springer, to appear.
8) M. Gordon and T. Melham. *Introduction to HOL*. Cambrige University Press, 1993.
9) C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
10) Y. Isobe and M. Roggenbach. Webpage on Csp-Prover. `http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`.
11) F. S. E. Limited. Failures-divergence refinement: FDR2. `http://www.fsel.com/`.
12) T. Nipkow, L. C. Paulon, and M. Wenzel. *Isabelle/HOL*. LNCS 2283. Springer, 2002.
13) S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *CADE'92*, LNAI 607, pages 748–752. Springer, 1992.
14) L. C. Paulson. *A Generic Theorem Prover*. LNCS 828. Springer, 1994.
15) I.P. RixGroenboom, ChrisHendriks. Algebraic proof assistants in HOL. In B. Möller, editor, *MPC'95*, LNCS 947, pages 305–321. Springer, 1995.
16) M. Roggenbach. CSP-Casl – A new integration of process algebra and algebraic specification. *Theoretical Computer Science*, to appear.
17) A. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
18) P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, 2001.
19) S. Schneider. Verifying authentication protocol implementations. In B. Jacobs and A. Rensink, editors, *FMOODS 2002*, *IFIP Conference Proceedings* Vol. 209, pages 5–24. Kluwer, 2002.
20) H. Tej. *HOL-CSP: Mechanised Formal Development of Concurrent Processes*. BISS Monograph Vol. 19. Logos Verlag Berlin, 2003.
21) H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J.Fitzgerald, C.B. Jones, and P.Lucas, editors, *FME'97*, LNCS 1313, pages 318–337. Springer, 1997.