TR-0899

代数的仕様を用いたソフトウェア開発支援環境 Metis-AS

大須賀　明彦（東芝）、坂井　公（筑波大）、
本位田　真一（東芝）

November, 1994

© Copyright 1994-11-16 ICOT, JAPAN　ALL RIGHTS RESERVED

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191～5

Institute for New Generation Computer Technology

代数的仕様を用いたソフトウェア開発支援環境 Metis-AS

大須賀 昭彦[†]　　坂井 公[‡]　　本位田 真一[†]

概　要

Metis-AS は代数的な仕様記述言語を用いたソフトウェア開発の支援を目的として，項書換え技術の研究環境 *Metis* に多ソート代数やモジュール化の概念を導入して実現されたシステムである．このシステムの特徴は，(1) 柔軟なユーザ・インタフェースによって仕様の記述や検証を効率的にすすめる，(2) いくつかの形式的意味論によって広範囲の記述段階を支援する，(3) 項書換え技術に基づく強力な仕様解析・検証機能を有することなどである．本論文では，*Metis-AS* の特徴的な機能を紹介し，それらの機能がソフトウェアの仕様化をいかに支援するかについて述べる．

1　はじめに

代数的仕様記述法 (algebraic specification techniques) は 1970 年代に抽象データ型のための形式的記述法として提案され，その後ソフトウェアの形式的仕様記述法として活発に研究開発がすすめられている．その特徴の一つは，仕様記述を等式論理と呼ばれる論理モデルによって宣言的に解釈できると同時に，項書換え系 (term rewriting system) と呼ばれる計算モデルによって手続き的にも解釈できることである．これにより，宣言的解釈に基づく仕様の厳密な検証や，手続き的解釈に基づく仕様の実行，または仕様から実現への変換などが可能となり，他の記述法と比較しても多くの利点を持つ[10]．

形式的仕様記述を用いたソフトウェア開発を実用化するためには，開発支援環境や開発方法論，ドキュメント，教育体系などの整備が必要と言われているが，代数的仕様記述法に関する前述の利点を活かすためには，支援環境の存在が特に重要である．この種の開発支援環境には，記述性に優れた言語と適切な意味論を備え，定理証明や仕様実行，静的解析，プログラム変換などの機能をバランスよく有することが望まれるが，既存の代表的な言語と処理系を概観してもこれに十分に応える環境，特に定理証明を効果的に支援するものが見当らない[5]．

そこで我々は，項書換え技術の研究環境 *Metis* [13] に多ソート代数やパラメータ化モジュールの概念を導入し，代数的仕様記述の支援環境 *Metis-AS* を構築した．その目的は，実システムの開発へこの環境を適用し，有効な支援技術やその効率的な実現法を明らかにすることである．このため *Metis-AS* は，(1) 柔軟なユーザ・インタフェースによって仕様の記述や検証を効率的にすすめる，(2) いくつかの形式的意味論によって広範囲の記述段階を支援する，(3) 項書換え技術に基づく強力な仕様解析・検証機能を有するなどの特徴を持つ．この中で，検証機能に関連する定理証明手続きの理論や応用については文献 13, 14, 15) などで既に報告されている．本論文では，各種手続きを有効に活用するための仕様記述言語の設計や，ソフトウェア開発支援の立場でのユーザ・インターフェースや推論技術の特徴などに焦点をあて，これらの技術がソフトウェアの仕様化をいかに支援するかを明らかにする．

まず第2章で *Metis-AS* の概要を紹介し，第3 章では仕様記述言語について述べる．*Metis-AS* の特徴を第4章で述べ，第5章では *Metis-AS* がソフトウェアの仕様化をいかに支援するかについて例を用いて示す．最後に第6章で関連システムとの比較を行う．

2　Metis-AS の概要

Metis-AS は項書換え技術の研究環境 *Metis* [13] に多ソート代数やモジュール化の概念を導入した拡張システムである．第 5 世代コンピュータプロジェクト (FGCS) において開発がすすめられ，これまでにSUN ワークステーション上のProlog 版の他に，並列推論マシン PIM [16] 上に並列検証を実装した *Metis-AS*/PIM が実現されている．また *Metis-AS*/PIM は，知的プ

† (株) 東芝 システム・ソフトウェア生産技術研究所
‡ 筑波大学 電子・情報工学系

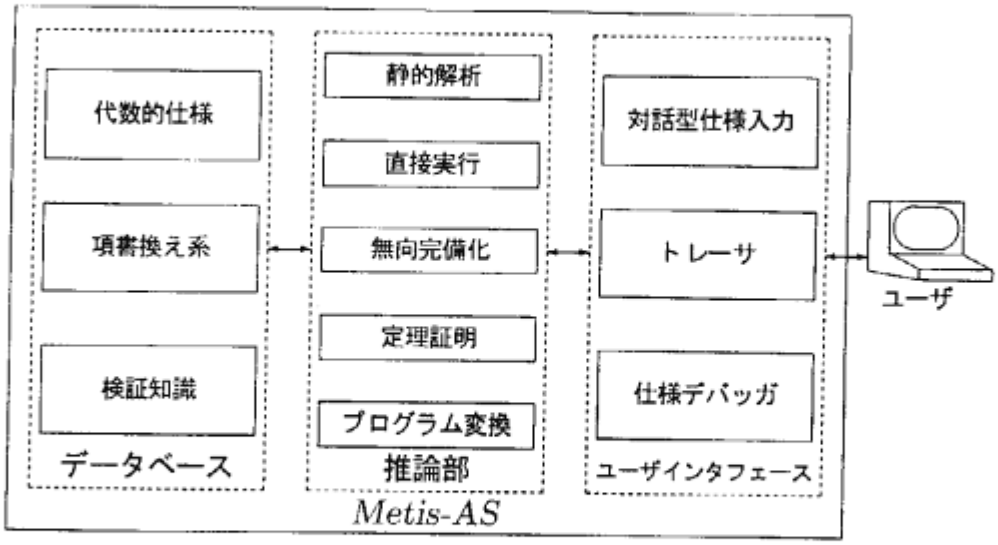

図 1　*Metis-AS* のシステム構成

ログラミング環境 MENDELS ZONE [9] の検証支援サブシステムとしても稼働している．

Metis-AS のシステム構成を図 1 に示す．データベース部には，代数的仕様，仕様から獲得される項書換え系，仕様検証を補助する知識などが蓄えられる．推論部は，仕様の静的な解析を行う静的解析機能，仕様を実行する直接実行機能，仕様から曖昧性や矛盾を取り除く無向完備化，仕様の性質を検証するための定理証明，仕様をプログラムへ変換するプログラム変換などからなる．ユーザ・インタフェース部には，ユーザとの対話によって仕様を構築する対話型仕様入力部，システムの推論状況や内部データを見やすい形で表示するトレーサ，仕様をデバッグするための仕様デバッガなどが用意されている．

3　Metis-AS における仕様記述言語

Metis-AS の仕様記述言語は，多くの代数的言語と同様に，多ソート代数に論理的な基礎をおく．このため，言語の基本的な構成要素はソート名の定義 S，各ソート上の関数記号 (演算子 $\mathcal{D}$ と構成子 C) の定義，記述に用いる変数名の定義 $\mathcal{V}$，そして公理の定義 Eからなる．これらを組み合せたものを基本仕様と呼び，次の 5 組 $BSP = (S, \mathcal{D}, C, \mathcal{V}, E)$ で表す．また，定理 Tや補題 L，定理証明のためのメタ推論規則 $\mathcal{M}$ などは検証に必要な記述なので，これらを組み合せたものを検証仕様と呼び，3 組 $VSP = (T, L, \mathcal{M})$ で表す．*Metis-AS* における代数的仕様は，基本仕様と検証仕様の組 $SPC = (BSP, VSP)$ である．これを記述する言語の概要を図 2 に示し，詳細な構文規則を付録 1 に載せる．以下で，言語の特徴について簡単に述べる．

```
spec モジュール名 with パラメータ定義 has
    uses          参照定義
    sorts         ソート名定義
    functions     演算子定義
    constructors  構成子定義
    forall        変数定義
    axioms        公理定義
    theorems      定理定義
    lemmas        補題定義
    ##            メタ規則定義
end.
```

図 2　仕様記述言語の概要

- 言語の特徴

言語の大枠は通常の代数的言語と同様であるが，次の点に特徴がある．それらは，(1) 代数構造を明確に定義するために関数記号を演算子と構成子に分類している，(2) 検証に関する情報を仕様中に数多く記述できる，(3) 記述性を向上させるための強力なマクロ展開機能を持つ，(4) 記述の抽象度に応じた複数の意味論を持つ，(5) 検証性のよさを意識したモジュール定義/参照法を採用しているなどである．また，通常の言語と比べて機能が不足している点は，順序ソートを支援しないこと，条件つき等式の記述を許さないことなど，主に記述性に関するものである．これらは定

理証明の効率化のために言語仕様からはずしたものであるが，マクロ展開機能などがこれを補っている．本章では特徴の (1) ～ (3) について述べ，(4) ～ (5) は *Metis-AS* の特徴に絡めて次章で説明する．

• 演算子と構成子の分類

関数記号は文献 15) の意味での演算子 $\mathcal{D}$ と (自由) 構成子 $\mathcal{C}$ に分けて記述される．たとえば，自然数の加算を普通に仕様化した場合，+は演算子に，0 と suc (後者関数) は構成子に分類される．これを代数モデルの上で考えると，演算子は代数構造上の演算を定義し，構成子は代数構造の要素を定義するものとなる．*Metis-AS* を用いた仕様記述では，初期の段階でソート名とその上の演算の種類を定義することによって問題を分析し，設計がすすむにつれて構成子を用いて各ソートの構造を明確にし，演算を帰納的関数として厳密に定義していくことになる．最終的に構成子が定義された仕様においては，文献 14) の意味での仕様の帰納的な完全性を仮定する．つまり，以下の条件が満足されるものとする．

(1) 任意の基礎項 $t \in T(\mathcal{D} \cup \mathcal{C})$ について $t \stackrel{\cdot}{=}_E u$ なる基礎構成子項 $u \in T(\mathcal{C})$ が存在する．

(2) 任意の基礎構成子項 $t, u \in T(\mathcal{C})$ について $t \stackrel{\cdot}{=}_E u$ となるのは $t \equiv u$ のときに限られる．

ここで，$T(\mathcal{D} \cup \mathcal{C})$ は $\mathcal{D}$ と $\mathcal{C}$ のみから構成される (つまり変数を持たない) 項の集合で，$T(\mathcal{C})$ は $\mathcal{C}$ のみから構成される項の集合を表す．また，$\stackrel{\cdot}{=}_E$ は公理 E による合同関係，$t \equiv u$ は 2 項 t, u の構文的な一致を表す．この条件は，演算子を含む基礎項が適当な順序のもとで必ず構成子項まで簡約できること (もし 0+suc(0) などが簡約できなければ誤り) と，意味的に等しい基礎構成子は構文的にも一致すること (もし 0=suc(0) などとなったら誤り) を保証する．これにより仕様記述における構文と意味の関係が明確になり，仕様の解析や検証が見通しよいものとなる．

• 検証情報の記述

仕様に記述される定理 T は，定理証明を行う際にそのままゴールとして扱われる．また補題 L には，T の証明を補助する性質が等式として記述される．L は証明途中で定理を簡約する単純化規則として用いられるので，L によって証明の負荷を増大させずに推論を効率化できる．証明の効率化のためにいかに適切な補題を獲得するかは重要かつ難しい問題であるが，*Metis-AS* では定理に依存する補題を必要に応じて会話的に獲得することができるので，汎用的な補題のみを L に記述しておけばよい．たとえば，`&`,`!`,`~`,`if` をそれぞれ論理積，論理和，否定，if 文の演算子とするとき，以下は汎用的な補題の例である．

```
lemmas
    x&x==x; x&~x==false; x!x==x; x!~x==true;
    ~(~x)==x; if(c,y,y)==y;
```

厳密な保証のためには，これらの補題も定理として証明しておくことが望ましい．

• マクロ展開

マクロ展開は仕様の記述性を向上させるための機能で，システム組込みのものとユーザ定義のマクロが用意されている．組込みマクロで最も頻繁に使われるのは等号の展開である．代数的仕様においては，ほとんどのソートに要素の等価性を判定する等号 $=$ $(\in \mathcal{D})$ が必要となる．これを組込み関数としている多くの処理系では，$l = r$ の判定結果を l と r の簡約結果の一致で決定しているが，定理証明を行うためには等号も公理化されている必要がある．そこで *Metis-AS* では，マクロ展開によって等号を自動的に公理化する．たとえば，自然数 nat の構成子を 0 と suc で定義し，公理に

```
= is_equation_on nat;
```

と記述すると，この公理は以下のように展開される．

```
0=0 == true;
0=suc(y) == false;
suc(x)=0 == false;
suc(x)=suc(y) == x=y;
```

つまり，着目しているソートの任意の構成子 $c, d \in \mathcal{C}$ について，$c(x_1, \ldots, x_m) = d(y_1, \ldots, y_n)$ の値を $c \neq d$ なら false とし，$c = d$ なら $(m = n$ なので) 右辺を $x_1 = y_1 \& \cdots \& x_m = y_n$ なる式に展開する．この展開の妥当性は，仕様の帰納的完全性に関する仮定から保証されている．その他に，ソート毎の if 文を生成する組込みマクロも用意されている．ユーザ定義のマクロの例としては，

```
# (L==R when C) -> (if(C,L,R)==R).
```

によって条件つき等式に近い形の記述を導入したり，

```
# op(950,fx,if).
# op(949,xfx,then), op(948,xfx,else).
# (if C then T else F) -> if(C,T,F).
```

によって if then else 構文を導入するなどが可能である[1]

[1] op(950,fx,if) は if が 950 の優先順位を持つ前置記号であることを，op(949,xfx,then),op(948,xfx,else) は then,else がそれぞれ 949,948 の優先順位を持つ中置記号であることを表す．優先順位は数値の小さなものほど結合の度合が強くなる．

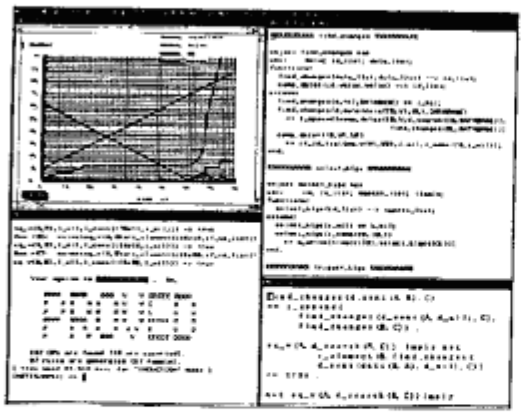

図 3　*Metis-AS*/PIM の画面例

4　Metis-AS の特徴

本章では *Metis-AS* 特徴的な機能の概略を述べる．なお，推論機能の詳細については，停止性の保証や無向完備化手続きに関して文献 13) を，定理証明手続きに関して文献 14, 15) などを参照されたい．

4.1　柔軟なユーザ・インタフェース

• 推論の実行モード

Metis-AS におけるすべての推論手続きは，自動/半自動/会話実行の 3 種類のモードを持つ．通常，新たな問題に推論を適用する場合には，まず会話実行モードによってステップ単位の推論を行い，公理や定理の修正，推論ステップの試行錯誤を繰り返しながら推論をすすめる．これによってある程度の見通しを得た段階で，今度は半自動実行モードによってユーザの判断が必要な分岐点まで自動推論を行う．最終的には自動実行モードによって効率よく推論をすすめ，検証の範囲を広げていく．会話実行モードにおいては，データの変更処理として，公理や定理の追加・削除，等式～書換え規則間のデータ変換，公理や定理に対する簡約やナローイングの実施などが行える．また，推論過程を制御する処理としては，推論の後戻り，次に処理される式の変更，帰納的定理証明における帰納項位置の変更などが行える．

• トレース表示

データの変更や推論過程の制御をうまく行うためには，推論状況をいかに把握するかが重要である．トレース表示はさまざまなレベルの推論情報をユーザに提供するための機能である．たとえば，推論の計算過程や構成された証明木の表示，特定の式に対する推論履歴の表示や木構造の表示，等式・書換え規則数のグラフ表示などを行うことができる．図 3に *Metis-AS*/PIM の画面例を示す．この画面においては，右上に仕様記述が，右下に証明中の定理が，左下には定理証明の進行の様子が示されている．また，左上のウィドウは定理証明の進行 (横軸) に伴って生成される各種データの累積数 (縦軸) をグラフ表示しており，証明の現状や以降の見通しについて，たとえば証明がどの程度のステップで停止するかなどを直観的に把握できる．

• 仕様デバッガ

定理証明によって矛盾が発見された場合，公理が複雑に作用して矛盾を生じていることも多く，原因の追求は容易な作業でない．そこで，仕様デバッガはユーザとの対話によって矛盾の原因の絞り込みを行う．その仕組みは，まず矛盾を含む証明木中のデータをユーザに提示し，ユーザが矛盾を含む枝の選択を繰り返すことで，最終的に誤りを含む公理に到達するというものである．

• 柔軟な解析と厳密な解析

Metis-AS は柔軟な解析と厳密な解析という 2 種類の解析/入力モードを持つ．柔軟なモードは，主に仕様記述の初期段階において手軽な操作によって仕様の実行や完備化，定理証明などを試し，厳密な仕様の下地を作るためのモードである．このモードでは，公理や定理，関数記号などの情報を最初からすべて与える必要はなく，ユーザは段階的に情報を追加・削除していくことができる．また，型推論などによって補うことのできる情報は自動的に追加される．たとえば，ユーザが最初に等式を入力した場合，そこに出現する関数記号が単一ソート上の関数としてとりあえず登録され，関数の型やソート名が明らかになった時点で元の情報が修正される．この情報は代数的仕様の構文形式に従ってファイルへ出力することも可能で，その場合には，ユーザがファイルの内容に手を加えて仕様を完成させることもできる．柔軟なモードのままで推論を実行する際には，推論を行うために必要な前提条件が厳しく検査されるので，推論結果の妥当性は常に保証される．

4.2　仕様記述の各段階を支援する意味論

代数的言語の代表的な意味論には，始代数意味論，終代数意味論，古典的 (曖昧) 意味論，操作的意味論などがある [4, 10]．この中で，古典的意味論はすべての可能な代数モデルのクラスを考える意味論であり，同型なモデルを同一視しても始代数モデルのような

一意性を持たない曖昧なモデルである．しかし，このモデルでは詳細に触れずに機能の本質を扱うことができるので，仕様記述の初期段階で抽象的な要求を記述するのには適したモデルとも言える．一方，仕様の詳細化が進んだ段階では代数構造やその要素が明らかになり，演算の意味も明確に捉えられるようになるので，これを表現できる始代数意味論が適当となる．また，最終的な仕様はプログラムに変換されるて実行されるので，操作的な意味論が適当である．そこで *Metis-AS* では，終代数意味論を除いた 3 つの意味論について，仕様記述の初期段階には古典的意味論を，設計段階には始代数意味論を，そして最終的な仕様には項書換え系に基づく操作的意味論を適用する．

4.3 項書換え技術に基づく種々の推論

- **静的解析**

代数的仕様は強く型づけされた言語なので，厳密な解析モードで入力された仕様に対しては，関数記号や変数宣言の不備，関数や変数の不適切な箇所への適用，公理化されていない演算子や公理中で用いられない構成子の存在，記述から型を決定できない曖昧な多重定義 (overloading) の使用などを検査し，必要に応じて警告を発することができる．これによって単純なミスによる誤りの多くは排除され，実行時の型エラーも起こらない．この段階で，見かけ上は同一の名前を持った異る関数記号についてもシステム内では別々の内部名が対応づけられるので，以降の内部処理では関数の型を考慮する必要がなくなる．

- **書換え規則の停止性保証**

項書換え技術に基づく推論では，簡約の停止性保証が重要な問題となる．これについては，辞書式部分項順序，勝抜き順序，多重集合経路順序などの単純化順序 [3] を適用する．これらの順序法は関数記号毎に自動的に選択され，引数の比較順についても最適なものが設定される．これらの推論によっても等式の停止性を保証できない場合，無向書換え規則の導入や新たな関数記号の導入といった対処が手続き毎に行われる [13]．

- **無向完備化手続き**

無向完備化手続き (unfailing Knuth-Bendix) は，公理を無向完備な項書換え系へ変換するものである [2]．従来の完備化手続きは，(1) 等式を停止的な書換え規則に変換できない，(2) 書換え規則が無限に生成されるなどの場合に失敗に終っていたが，*Metis-AS* の無向完備化手続きでは，豊富な順序法や無向書換え規則の導入により (1) を完全に回避し，(2) に対しても無限生成を検出する評価基準によって回避しやすいように工夫されている [13]．さらには，不要な要対を獲得しないための評価基準も導入し，手続きの効率化をはかっている．また，仕様記述の帰納的完全性についての検査が手続き中で常に行われているので，関数定義の不完全性 (たとえば，0+suc(0) が簡約できない) や公理の意味的な矛盾 (たとえば，0=suc(0) が公理から導かれる) があれば必ず検出する．

- **仕様の直接実行**

仕様の直接実行は，公理を項書換え系とみなし，与えられた項を簡約することによって行われる．項書換え系は，入力した公理を単純に左から右への書換え規則とみなすものと，無向完備化によって得られる完備なものとを選択できる．*Metis-AS* における仕様は操作的な意味を意識せずに書かれるので，前者の実行には健全性 (つまり，得られた解は正しい) のみが保証される．一方，後者の実行は公理によって定まる等式理論に対して完全なものとなる．

- **定理証明手続き**

Metis-AS では，演繹的定理証明と帰納的定理証明の 2 つの手続きを用いて，仕様が定理に記述された性質を満足するかを検査する．従来のほとんどの定理証明が完備な項書換え系の存在を前提としていたのに対し，*Metis-AS* の証明手続きはこれを前提としないので，仮に前の段階で無向完備化が成功していなくても定理証明を適用できる点が特徴となっている [13, 15]．演繹的証明は古典的意味論の上で演算の抽象的な性質を証明するために用いられ，帰納的証明は始代数意味論の上で演算の詳細な性質を調べるために用いられる．

仕様が構成子を持たない段階で定理証明を起動すると演繹的証明が実行される．この手続きによって，たとえば群や環，体などの代数構造に関する推論や，半順序，全順序集合などの性質に関する検証が行える．演繹的定理証明の実行は特別な戦略を必要としないので，*Metis-AS* の自動実行モードによってユーザの介入なしに全自動実行が行える．

帰納的定理証明の実行は，多くの場合，問題に応じた適切な補題の獲得を必要とする．そこで，*Metis-AS* の自動実行モードはほとんどの手続きを自動的にすすめるが，補題の必要性が検出された場合のみユーザに判断を求める．これに対してユーザが適切な補題を与えると，元の定理の証明が完了した後に与えられた補題に対する証明が開始される．つまり，以下の流れで証明がすすめられる．

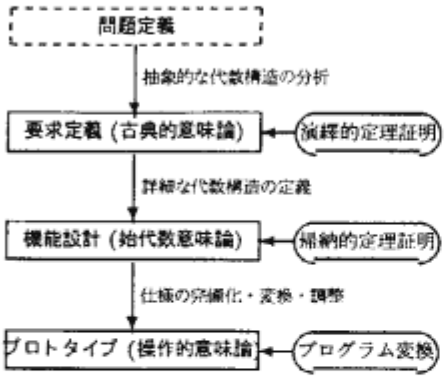

図4 *Metis-AS*に支援されたソフトウェア開発の流れ

(1) 帰納的定理の証明を開始する.
(2) 補題の必要性が検出されると，ユーザの判断に基づいて補題を獲得する.
(3) (2) を繰り返すことによって定理の証明を完了させる.
(4) 途中で獲得した補題を新たな帰納的定理に位置づけて，(1) の処理へ戻る.

ユーザが補題を与える際には補題の健全性検査が行われる．これは，入力された補題中の変数に一定の深さまでの基礎構成子項を網羅的に代入し，すべての場合に補題の両辺が同一項へ簡約されるかを調べるものである．ユーザが誤った補題を与えた場合，ほとんどの誤りがこの検査によって発見される.

*Metis-AS*における仕様記述の段階と適用される意味論，それを支援する推論技術について図4にまとめる.

• **モジュールの検証**

Metis-AS の仕様はモジュール単位で記述される．定義済みのモジュールを参照する際には，パラメータの指定または名前の変更によってモジュールの内容を具体化もしくは変更して利用できる．たとえば，任意のソートを要素とするリスト構造の仕様は，以下のように記述できる.

```
spec LIST with [elm/X] has
    uses BOOL protecting;
    srts assume X,list;
    cons
       []:-->list;
       [_|_]:X,list-->list;
```

ここで予約語 assume は，ソート X の内容が後から定義されるものであることを表明している．このモジュールに定理証明を適用すると，X の構造に依存しないリストの帰納的性質などは証明されるが，X の要素に関わる性質については X の構成子が十分に定義されるまで証明が保留される．また，モジュールを利用する側で定理証明を行う際には，基本的にすべてのモジュール内容を展開して推論を行うが，protecting 指定によって参照されるモジュールについては内容の変更があり得ないので，既に証明済みの性質や獲得済みの項書換え系はそのまま継承される.

• **プログラム変換**

プログラム変換は仕様から得られた項書換え系を等価なプログラムへ変換する．項書換え系からプログラムへの変換手続きは既に多くのものが提案されているが，*Metis-AS* (Prolog 版) は Prolog プログラムを生成し，*Metis-AS*/PIM は KL1 プログラム [17] を生成する．一般に，仕様から変換されたプログラムでは実行の効率が問題となるが，*Metis-AS* の最終的な仕様は代数モデル上の帰納的関数を定義しているため，これを機械的に変換したプログラムにおいても効率はさほど問題にならない．また，*Metis-AS*/PIM によって得られる KL1 プログラムは並列簡約をシミュレートして PIM 上で並列実行されるため，Prolog プログラムに比べてさらに効率化されている.

• **メタ推論による証明能力の向上**

Metis-AS では推論能力を向上させるためのメタ推論規則 $\mathcal{M}$ をユーザが定義できる．たとえば，LP [7] でシステム組込みとなっている if 文のメタ変換，つまり $\mathrm{if}(c, t[c], f[c])$ なる項を $\mathrm{if}(c, t[\mathrm{true}], f[\mathrm{false}])$ に変換する処理は，以下のように記述できる[2].

```
## if(C,T,F)==R ###> [ if(C,T1,F1)==R ]
where replace(C,true,T,T1),replace(C,false,F,F1);
```

ここで，replace(C,true,T,T1) は項 T における C のすべての出現を true に置き換え，その結果を T1 に返すプログラムの呼びだしである．また，x&y=true なる等式を x=true と y=true の 2 つの等式に分解するなどの処理も以下のように簡単に定義できる.

```
## X&Y==true ###> [ X==true, Y==true ];
```

また，帰納的定理証明において補題による単純化を行う際，関数記号に AC (結合律・交換律)，C (交換律) などの属性が定義されていると，属性つきパターンマッチを用いた簡約が適用される．たとえば，補題として $-x+x=0$ が定義され，+の属性が

```
## AC [ + ];
```

のように宣言されていると，$a+b+(-a)+c$ などの式を $b+c$ へ簡約することができる.

この種のメタ推論は実用的な定理証明を実現する上で不可欠と考えられるので，ユーザがこれを定義し

[2] $t[c]$ を $t[\mathrm{true}]$ に変換するとは，項 t における c の出現を true に置き換えることを意味する.

て，メタ推論を含めた試行錯誤を行える点は *Metis-AS* の大きな特徴となっている．ただし，メタ推論規則そのものの正しさを定理証明の枠内で確認することは難しいので，$\mathcal{M}$ を定義する際にはその妥当性に細心の注意を払う必要がある．

- **各機能のねらい**

一般に，ソフトウェアの仕様記述の正しさについて議論する際には，

(1) 記述に抜けがない (完全性)，

(2) 記述に矛盾がない (無矛盾性)，

(3) 仕様が要求に適合している (仕様の正しさ)，

(4) 実現が仕様に適合している (実現の正しさ)

などが問題となる．これらの項目に対して，本章で述べた *Metis-AS* の主要機能が保証を与えようと目指している範囲を表 1に示す．この表において，○で示した項目は全体的な保証を，△は部分的な保証を与えようとするものである．完全性の保証が常に部分的であるのは，ユーザが表明していない潜在的な要求に対する記述の抜けを検証することが原理的に不可能なためである．逆に，不完全な要求であっても記述さえしてあれば，これに対して完備化や定理証明を適用し，要求項目自体を修正しながら検証をすすめることができる．

5 ソフトウェア仕様記述の例

本章では，*Metis-AS* がいかにソフトウェア仕様の記述を支援するかを簡単な例題によって見る．例題としては，「エレベータ内ボタン操作」[15] を用いる．図 5に仕様の記述例を示す．この仕様において重要な性質は，(1) 初期状態ではどの階に対する停止指示もない，(2) ボタンを押さない階に対する指示は変化しない，(3) ボタンを押した階に対する指示は停止であるなどであり，これらは既に定理として記述されている．*Metis-AS* による支援をわかりやすく説明するために，この仕様はいくつかの誤りを含んでいる．それらは，(e1) 関数に与える引数順の誤り，(e2) 使用されない構成子の存在，(e3) 記述の矛盾，(e4) 必要な公理の不足，そして (e5) 記述の意味的な誤りの 5 つである．

表 1 *Metis-AS* の主要機能とそのねらい

機能 \ 項目	完全性	無矛盾性	仕様正しさ	実現正しさ
静的解析	△	△		
直接実行	△		△	
無向完備化	△	○		
定理証明	△	○	○	
プログラム変換				○

```
spec エレベータ内ボタン操作 has
    use BOOL protecting;
    sorts 階番号, 停止階情報, 動作指示;
    functions
     ボタン押下げ:階番号, 停止階情報-->停止階情報;
     動作指示参照:階番号, 停止階情報-->動作指示;
     = :階番号, 階番号-->bool;
     if _ then _ else
        :bool, 停止階情報, 停止階情報-->停止階情報;
     if _ then _ else
        :bool, 動作指示, 動作指示-->動作指示;
    constructors
     InitStop:-->停止階情報;
     AddStop:階番号, 停止階情報-->停止階情報;
     F1,...,F5:-->階番号;
     停止, 通過:-->動作指示;
     Amb:-->停止階情報;
          /*** (e2) 使用されない構成子 ***/
    forall
     f,...,f3:階番号;
     s:停止階情報;
    axioms /*** (e4) ボタン押下げ公理の不足 ***/
     ボタン押下げ(f1,AddStop(f2,s))
      ==if f1=f2 then AddStop(f2,s)
         else AddStop(ボタン押下げ(f1,s),f2);
          /*** (e1) 引数順の誤り ***/
     動作指示参照(f,InitStop)==停止;
          /*** (e5) 意味的な誤り ***/
     動作指示参照(f1,AddStop(f2,s))
      ==if f1=f2 then 停止 else 動作指示参照(f1,s);
     動作指示参照(f1,AddStop(f2,s))
      == if f1=f2 then 通過 else 動作指示参照(f1,s);
          /*** (e3) 記述の矛盾 ***/
     = is_equation_on 階番号;
     if is_if_then_else_on 停止階情報;
     if is_if_then_else_on 動作指示;
    theorems
     動作指示参照(f,InitStop)==通過;
     動作指示参照(f1, ボタン押下げ(f2,s))
      ==動作指示参照(f1,s) when ~(f1=f2);
     動作指示参照(f, ボタン押下げ(f,s))==停止;
end.
```

図 5 エレベータ内ボタン操作の記述例

- **静的解析**

静的解析によって最初に発見されるのは，(e1) のような構文的誤りである．厳密な解析モードでこの仕様を読み込むと，この誤りはただちに発見される．(e1) を修正すると，次に (e2) の使用されない構成子についての警告が発せられる．

- **無向完備化による検査**

仕様の無向完備化を行うと，図 6に示すように仕様の矛盾が発見される．その際，仕様デバッガによって，矛盾を引き起こす原因となっている公理が表示されるので，ユーザはこの情報から仕様の矛盾をさがしだすことができる．この例では，誤り (e3) の記述

```
[METIS-AS]-> ukb  ..... 仕様の完備化指示
[ unfailing completion for "エレベータ内ボタン操作" ]
New ボタン押下げ(A,AddStop(B,C))==>
if(A=B,AddStop(B,C),AddStop(B, ボタン押下げ(A,C)))
New 動作指示参照(A,InitStop)==>停止
New 動作指示参照(A,AddStop(B,C))==>if(A=B, 停止, 動作指
示参照(A,C))
New if(A=B, 停止, 動作指示参照(A,C))
                ==>if(A=B, 通過, 動作指示参照(A,C))
    ******* INCONSISTENT *******  ..... 矛盾の発見
    By 通過 = 停止  ..... 矛盾内容の表示
You want to inspect ? (*y/n) yes
    ******* CHECK THE FOLLOWING *******
Axiom(s):
  動作指示参照(A,AddStop(B,C))==
              if(A=B, 停止, 動作指示参照(A,C))
  動作指示参照(A,AddStop(B,C))==
              if(A=B, 通過, 動作指示参照(A,C))
          ..... 矛盾を引き起こした公理の表示
```

図 6　無向完備化の実行 (1)

```
[METIS-AS]-> uk
[ unfailing completion for "エレベータ内ボタン操作" ]
New ボタン押下げ(A,AddStop(B,C))==>
  if(A=B,AddStop(B,C),AddStop(B, ボタン押下げ(A,C)))
New 動作指示参照(A,InitStop)==>停止
New 動作指示参照(A,AddStop(B,C))==>if(A=B, 停止, 動作指
示参照(A,C))
**<ERR>** function
   "ボタン押下げ(F1,InitStop)" is not reducible.
**<ERR>** function
   "ボタン押下げ(F2,InitStop)" is not reducible.
**<ERR>** function
   "ボタン押下げ(F3,InitStop)" is not reducible.
**<ERR>** function
   "ボタン押下げ(F4,InitStop)" is not reducible.
**<ERR>** function
   "ボタン押下げ(F5,InitStop)" is not reducible.
          ..... 関数定義の不完全性の表示
```

図 7　無向完備化の実行 (2)

が表示中に含まれているので，これを削除することになる．修正した仕様を再度完備化すると，今度は図7のように関数定義の不完全性が発見される．これらのエラー表示は，ボタン押下げ関数が「ボタン押下げ(_,InitStop)」という入力に対して未定義であることを例示している．これにより，この演算が停止階情報の初期状態 InitStop に対して定義されていないことが明らかとなり，(e4) の場所に，「ボタン押下げ(f,InitStop)==AddStop(f,InitStop)」なる公理を追加することになる．この後に仕様を完備化すると，今度は矛盾も不完全性も発見されずに手続きが終了する．

- 仕様の直接実行による動作確認

完備な項書換え系を用いて仕様を直接実行した様子を図8に示す．まず，初期状態で3階のボタンを押し，次に5階のボタンを押す操作をシミュレートする．ここで，「state:=t」なる記述は t の簡約結果を状態変数 state へ代入する指定，「@state」は state へ代入されている値を参照する指定，「@」は1つ前の簡約

```
Term> state:=ボタン押下げ(F3,InitStop).
=*=> AddStop(F3,InitStop)  ..... 実行結果の表示
state = AddStop(F3,InitStop)  ..... 停止階情報の表示

Term> state:=ボタン押下げ(F5,@state).
=*=> AddStop(F3,AddStop(F5,InitStop))
state = AddStop(F3,AddStop(F5,InitStop))

Term> state:=ボタン押下げ(F3,@).
=*=> AddStop(F3,AddStop(F5,InitStop))
state = AddStop(F3,AddStop(F5,InitStop))

Term> 動作指示参照(F5,@).
=*=> 停止
state = AddStop(F3,AddStop(F5,InitStop))
```

図 8　簡約の実行

```
[METIS-AS]-> pro  ..... 自動検証の指示
[inductive theorem proving for "エレベータ内ボタン操作"]
Prove 動作指示参照(A,InitStop)==通過 (*y/n) ? y
Prove if(~(A=B), 動作指示参照(A, ボタン押下げ(B,C)), 動作
指示参照(A,C))
 ==動作指示参照(A,C) (*y/n) ? y
Prove 動作指示参照(A, ボタン押下げ(A,B))==停止 (*y/n) ? y
    ******* DISPROVED *******
    停止 = 通過 (theorem)
You want to inspect ? (*y/n)
    ******* CHECK THE FOLLOWING *******
Target(s):
  動作指示参照(A,InitStop) == 通過
Axiom(s):
  動作指示参照(A,InitStop) == 停止
T(heorems)    : 3 generated, 3 remain.
S(implifiers): 0 generated.
Narrow(s)     : 0 found, 0 asserted.
Reduction     : 1 steps.
Runtime       : 40 msec
              (  0% for selection,  0% for narrowing,
                 0% for ordering, 0% for T-reduction,
                 100% for others )
```

図 9　帰納的定理証明の実行

結果を参照する指定である．この実行結果から，3階，5階が共に停止階情報に登録されていることが確認できる．もう一度3階のボタンを押しても状態は変化せず，さらに5階に関する動作指示を参照すると停止の指示が正しく返され，これだけの実行では最後の誤り (e5) を発見することができない．これは，プログラムのデバッグにおいてテストケースの不足によってバグが見逃されるのと同じ状況と言える．

- 帰納的定理証明による検証

帰納的定理証明によって3つの定理の証明を試みると，図9のように定理の不成立が告げられる．矛盾の原因となっている公理をさがすために仕様デバッガを起動すると，誤った記述 (e5) が原因の候補として表示される．これを修正した仕様によって定理を証明すると，今度は帰納的定理の成立が確認される．

- Prolog プログラムへの変換

こうして正しさを保証された仕様はプログラムへ

```
ボタン押下げ(A,InitStop,PushStop(A,InitStop)) :- ! .
ボタン押下げ(A,PushStop(B,C),D) :- !,
    =(A,B,E),
    ボタン押下げ(A,C,F),
    if(E,PushStop(B,C),PushStop(B,F),D) .
ボタン押下げ(A,B,ボタン押下げ(A,B)) :- ! .
動作指示参照(A,InitStop,通過) :- ! .
動作指示参照(A,PushStop(B,C),D) :- !,
    =(A,B,E),
    動作指示参照(A,C,F),
    if(E,停止,F,D) .
動作指示参照(A,B,動作指示参照(A,B)) :- ! .
```

図 10　変換によって得られたPrologプログラム例

変換されてプロトタイプとなる.「エレベータ内ボタン操作」の仕様をPrologプログラムへ変換した結果を図10示す.

6　比較と評価

Metis-AS の仕様記述言語とその支援環境を既存の言語や処理系と比較する．まず，代表的な代数的仕様記述言語と比べてみると，OBJ2，OBJ3，CLEAR，ACT ONE，ASL などには十分な定理証明機能が備わっていない[5]．これは，検証性よりも記述性を重視した立場の違いによるものと考えられる．2OBJ[8]は定理証明のための枠組を提供しているが，帰納的定理の証明機能が十分とは言えない．Larch を支援する定理証明系 LP は実用的な対話型証明機能を有するが，Larch 共通言語 (LSL) は代数的仕様において重要な役割を果たす始代数意味論を採用していない[7]．

一方，項書換え技術を核とする支援環境には TRSPEC[1]の証明系である UNICOM や REVE[12]，RRL[11]など，定理証明を支援するものが数多く存在する．しかし，ほとんどが実験的なシステムで，たとえばモジュール構造がなかったりパラメータ化ができないなど，仕様記述言語としての機能が不十分である．これに対して *Metis-AS* では，検証機能を備えたモジュール機構などを提供している．たとえば，仮パラメータに関して制約が存在する場合，これを定理として記述しておけば，モジュール利用の際に定理証明を用いて実パラメータの妥当性を検査できる．これは，OBJ2 や OBJ3 における理論 (theory) 記述[6]を検証つきで実現したものと見ることができる．

また，多くの代数的言語は操作的意味論を採用しているので，操作的な解釈に曖昧性が生じないよう，公理の記述形式が左線形 (left linear) かつ無重複 (non-overlapping) などに制限されることが多い．これに対して，*Metis-AS* の公理は宣言的に解釈されるので，記述形式の制限を必要としない．一方，操作的意味論を採用する多くの言語は，AC (結合律・交換律を法とする) パターンマッチなどの属性つきパターンマッチによって仕様の記述性を高めている．しかし，この種の属性を宣言的に扱うためには，AC 単一化などの拡張単一化だけでなく，停止性順序の拡張や完備化手続きの拡張なども必要で，これらは推論の効率を大きく後退させる要因となる．そこで *Metis-AS* では，補題を用いて定理を単純化するときにのみこの種のパターンマッチを行い，推論の効率を落さない範囲での支援を行っている．

7　おわりに

本論文では，代数的仕様を用いたソフトウェア開発支援環境 *Metis-AS* の機能や特徴について述べた．現在，我々は *Metis-AS* を用いた実システムの記述・検証実験に取り組んでいる．*Metis-AS* の仕様記述言語は検証性の良さを重視して設計されたものであるため，実適用の場面で記述性が問題となることも考えられる．現在の言語について記述性を向上させる項目としては，

(1) 順序ソートへの拡張，
(2) 条件つき等式 (公理) の導入，
(3) より柔軟な継承機能の実現，
(4) 例外処理記述方法の導入，
(5) 状態概念 (副作用) の導入，
(6) 属性つき単一化/定理証明の導入

などが挙げられる．今後は，適用実験から得られる言語拡張への要求と，それに伴う推論技術の適用性や効率の変化を調整しながら，*Metis-AS* を記述性と検証性のバランスのとれた言語と支援環境へ拡張していく予定である．

謝辞　本研究は，第5世代コンピュータプロジェクト (FGCS) の一環として行われたものである．研究の機会をいただいた，ICOT 渕一博 所長 (現在，東京大学教授)，長谷川隆三 部長 (現在，ICOT 次長)，(株) 東芝 システム・ソフトウェア生産技術研究所 渡辺貞一 所長，松村一夫 部長に感謝致します．

参考文献

1) Avenhaus, J. et al.: TRSPEC: A Term Rewriting Based System for Algebraic Specifications, *LNCS 308*, pp. 245–248, Springer-Verlag (1988).

2) Bachmair, L.: *Canonical Equational Proofs*, p. 135, Birkhäuser (1991).

3) Dershowitz, N. and Jouannaud, J.-P.: Rewrite Systems, in *Handbook of Theor. Comput. Sci.*, Vol. B,Chap. 6, pp. 245 320, MIT Press/Elsevier (1990).

4) Ehrig, H. and Mahr, B.: *Fundamentals of Algebraic Specification 1 & 2*, EATCS Monographs on Theor. Comput. Sci., Vol. 6&21, Springer-Verlag (1985,1990).

5) Ehrig, H.: Overview of Algebraic Specification Languages, Environments and Tools, and Algebraic Specifications of Software Systems, *Bull. EATCS 39&40* (1989).

6) Futatsugi, K. et al.: Principles of OBJ2, in *Proc. 12th ACM Symposium on Principles of Programming Languages*, pp. 52–66 (1985).

7) Garland, S. J. and Guttag, J. V.: An Overview of LP, The Larch Prover, *LNCS 355*, pp. 137–151, Springer-Verlag (1989).

8) Goguen, J. et al.: 2OBJ: A metalogical framework theorem prover based on equational logic, *Phil. Trans. R. Soc. Lond.*, Vol. 339, pp. 69–86 (1992).

9) Houiden, S., Ohsuga, A., and Uchihira, N.: An integration environment to put formal specifications into practical use in real-time systems, in *Proc. 6th Int. Workshop on Software Specification and Design*, pp. 102–109 (1991).

10) 稲垣康善, 坂部俊樹: 抽象データタイプの代数的仕様記述法の基礎 (1)〜(4), 情報処理, Vol. 25, No. 1&5&7&9 (1984).

11) Kapur, D. and Zhang, H.: RRL: A Rewrite Rule Laboratory, *LNCS 310*, pp. 768–769, Springer-Verlag (1988).

12) Lescanne, P.: REVE: A Rewrite Rule Laboratory, *LNCS 230*, pp. 695–696, Springer-Verlag (1986).

13) Ohsuga, A. and Sakai, K.: Metis: A Term Rewriting System Generator, in *Software Science and Engineering*, pp. 1–15, World Scientific (1991).

14) 大須賀昭彦, 坂井公, 本位田真一: 等式論理の帰納的定理を証明する手続き, 信学論, Vol. J76-D-I, No. 3, pp. 130–138 (1993).

15) 大須賀昭彦, 坂井公, 本位田真一: Metis-AS における代数的仕様の検証手続き, 情報処理学会論文誌, Vol. 34, No. 11, pp. 2242–2250 (1993).

16) Taki, K.: Parallel Inference Machine PIM, in *Proc. FGCS'92*, pp. 50–72 (1992).

17) Ueda, K. and Chikayama, T.: Design of the Kernel Language for the Parallel Inference Machine, *The comput. J.*, Vol. 33, No. 6, pp. 494–500 (1990).

付録 1 (仕様記述言語の構文規則)

[x]	は x がオプションであることを表す.
{ x }*	は x の 0 回以上の繰返しを表す.
{ x // s }*	は s を区切り記号とする x の 0 回以上の繰返しを表す.
{ x // s }+	は s を区切り記号とする x の 1 回以上の繰返しを表す.

```
〈仕様〉::= { 〈全域マクロ定義〉| 〈オペレータ定義〉| 〈モジュール定義〉 }+
〈全域マクロ定義〉::= "#" { 〈マクロ宣言〉// "," }+ "."
〈マクロ宣言〉::= 〈非変数項〉"->" 〈項〉[ "where" 〈プログラム記述〉 ]
〈オペレータ定義〉::= "#" { 〈オペレータ宣言〉// "," }+ "."
〈オペレータ宣言〉::= "op(" 〈優先度〉"," 〈置形式〉"," 〈関数子〉")"
                    | "op(" 〈モジュール名〉")"
〈モジュール定義〉::=
    "spec" 〈モジュール名〉[ "with" 〈パラメータ定義〉] "has"
      { 〈局所マクロ定義〉 }*
      [ "uses" 〈参照定義〉]
      [ "sorts" 〈ソート名定義〉]
      [ "functions" 〈演算子定義〉]
      [ "constructors" 〈構成子定義〉]
      [ "forall" 〈変数定義〉]
      [ "axioms" 〈公理定義〉]
      [ "theorems" 〈定理定義〉]
      [ "lemmas" 〈補題定義〉]
      { 〈メタ規則定義〉 }*
    "end" "."
〈パラメータ定義〉::= "[" { 〈属性名〉"/" 〈属性値〉// "," }+ "]"
〈局所マクロ定義〉::= "#" { 〈マクロ宣言〉// "," }+ ";"
〈参照定義〉::= { 〈モジュール名〉[ "with" 〈パラメータ定義〉]
      [ "renaming" 〈変更定義〉] [ "protecting" ] ";" }+
〈変更定義〉::= "[" { 〈変更宣言〉// "," }+ "]"
〈変更宣言〉::= 〈ソート名〉"to" 〈ソート名〉
              | 〈関数名〉[ ":" 〈関数ソート定義〉] "to" 〈関数名〉
〈ソート名定義〉::= { [ "assume" ] 〈ソート名〉";" }+
〈演算子定義〉::= { 〈関数定義〉";" }+
〈構成子定義〉::= { 〈関数定義〉";" }+
〈関数定義〉::= [ "assume" ] { 〈関数名〉// "," }+ ":" 〈関数ソート定義〉
〈関数名〉::= 〈関数子〉| 〈添字付関数子〉"," "..." "," 〈添字付関数子〉
〈関数ソート定義〉::= [ { 〈入力ソート〉// "," }+ ] "-->" 〈ソート名〉
〈入力ソート〉::= 〈ソート名〉| 〈ソート名〉"*" 〈自然数〉
〈変数定義〉::= { { 〈変数名〉// "," }+ ":" 〈ソート名〉";" }+
〈変数名〉::= 〈変数子〉| 〈添字付変数子〉"," "..." "," 〈添字付変数子〉
〈公理定義〉::= { 〈等式〉";" | 〈補完等式〉";"
                | 〈限定等式〉";" | 〈等号演算子〉| 〈IF 演算子〉";"
〈定理定義〉::= { 〈等式〉";" | 〈端末入力指定〉";" }+
〈補題定義〉::= { 〈等式〉";" | 〈端末入力指定〉";" }+
〈等式〉::= [ 〈識別子〉":" ] 〈拡張項〉"==" 〈拡張項〉
〈端末入力指定〉::= "terminal"
〈補完等式〉::= "otherwise" 〈等式〉
〈限定等式〉::= 〈等式〉"foreach" 〈変数子〉
〈等号演算子〉::= 〈関数子〉"is_equation_on" 〈ソート名〉
〈IF 演算子〉::= 〈関数子〉"is_if_then_else_on" 〈ソート名〉
〈メタ規則定義〉::= "##" ( 〈メタ変換定義〉| 〈AC 関数定義〉) "."
〈メタ変換定義〉::= 〈等式〉"###>" "[" { 〈等式〉// "," }+ "]"
                    "where" 〈メタ変換プログラム〉
〈AC 関数定義〉::= ( "AC" | "C" ) "[" { 〈関数名〉// "," }+ "]"
〈拡張項〉::= 〈項〉| 〈拡張引数項を含む項〉' | 〈ソート指定項を含む項〉
〈拡張引数項〉::= 〈関数子〉"(" 〈項〉"," "..." "," 〈項〉")"
〈ソート指定項〉::= 〈項〉":" 〈ソート名〉
〈モジュール名〉, 〈属性名〉, 〈関数子〉, 〈変数子〉, 〈識別子〉::= 〈定数〉
〈属性値〉, 〈ソート名〉,::= 〈項〉
〈添字付関数子〉, 〈添字付変数子〉::= 〈定数〉〈自然数〉
〈優先度〉::= 〈0〜900 の整数〉
〈置形式〉::= "fx" | "fy" | "xfx" | "xfy" | "yfx" | "xf" | "yf"
```