

Title	PCTE を用いた UNIX コマンドデータベースの作成に関する研究
Author(s)	田中, 聡
Citation	
Issue Date	2000-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1352
Rights	
Description	Supervisor: 権藤 克彦, 情報科学研究科, 修士

修 士 論 文

PCTE を用いた UNIX コマンドデータベースの作成 に関する研究

指導教官 権藤 克彦 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

田中 聡

2000 年 2 月 15 日

目 次

1	はじめに	1
1.1	背景	1
1.1.1	ソフトウェアデータベースの必要性	1
1.1.2	UNIX ファイルシステムにおける制約・関係の記述の現状	2
1.2	研究概要	2
1.2.1	目的	2
1.2.2	アプローチ	2
1.2.3	成果	3
1.3	論文構成	3
2	UNIX コマンドの概要	5
2.1	UNIX のシステム	5
2.1.1	シェル	5
2.1.2	コマンド	6
2.2	実験材料としての UNIX コマンド	6
2.3	情報参照の現状と問題点	7
2.4	例題	9
3	PCTE	12
3.1	PCTE とは?	12
3.2	Emeraude PCIE	13
3.3	実験道具としての PCIE	14
3.4	SDS 図	14
3.4.1	ERA モデル	14
3.4.2	リンクカテゴリ	15

3.5	ツール	17
3.5.1	SDS で実現する制約と ツールにより実現する制約	17
3.5.2	ツール実現の為のライブラリ関数	19
4	UNIX コマンド の関連の記述	20
4.1	階層構造	20
4.1.1	メタ・クラス図	20
4.1.2	クラス図	21
4.1.3	インスタンス図	21
4.2	メタ・クラス図の作成	21
4.3	クラス図の作成	24
4.3.1	例	24
4.3.2	Emeraude PCTE 上での記述	27
4.4	インスタンス図の作成	28
4.4.1	自動作成の必要性和実現可能性	28
4.4.2	インスタンス図の自動作成	29
5	UNIXコマンド の制約の実現	32
5.1	機能	33
5.2	実現	34
5.3	評価	38
6	考察	40
6.1	評価	40
6.2	制約の分類	41
6.2.1	分類結果	41
6.2.2	C のプログラムとの比較	42
6.3	メタ情報	43
7	おわりに	45
7.1	まとめ	45
7.2	結論	46
7.3	今後の課題	47

第 1 章

はじめに

真のソフトウェアデータベースの実現の第一歩として、本研究では、対象を UNIX のファイルシステムに絞り、PCTE を用いてコマンドやファイル間の制約・関係の記述を試みた。

1.1 背景

1.1.1 ソフトウェアデータベースの必要性

近年、情報化社会の発展と共にコンピュータは急速に普及してきた。また、それと共にソフトウェアも急速に進歩・普及し、ソフトウェアの持つ情報は膨大なものとなった。ソフトウェアの膨大な情報の中から必要な情報を効率よく検索・参照できるためにはソフトウェアデータベースが必要である。ソフトウェアデータベースが必要になるのは大きく分けて以下の場合だと考える。

1. ソフトウェアを保守する。
2. 予期しない実行結果に対処する。
3. ソフトウェアの使用方法を知る。
4. 目的に応じたソフトウェアを探す。

ソフトウェアデータベースにおける必要な機能は各場合によって異なる。必要な機能はそれぞれ 1 制約・関係の記述の参照、2 動的情報¹の記述の参照、3 使用書やマニュアル

¹実行時まで決定されない情報。例として 環境変数の値やファイルの内部データ など。

の参照、4. 検索機能 である。

必要な ソフトウェアオブジェクトの制約・関係を 正確 かつ形式的に記述したソフトウェアデータベースは現在のところ存在しない。

UNIX コマンドを保守するためには UNIXコマンドにおける制約・関係の記述の参照が必要となり、UNIXコマンドの意図しない実行結果に対処するには、UNIXコマンドやファイル、環境変数などの具体的なオブジェクトを参照することが必要である。本研究ではその点に着目し、UNIXコマンドにおける制約・関係の記述と動的情報の記述という点に重点を置く。

1.1.2 UNIX ファイルシステムにおける制約・関係の記述の現状

現段階において、UNIXコマンドの情報を記述したものに、リファレンスマニュアル やオンラインマニュアル、ソースコード 等が存在する。しかし、オンラインマニュアル等は記述が形式的でなく、あいまいであるため検索が行ないにくい。また、ソースコードは形式的であるが、情報量が膨大であり、ソフトウェアデータベースとして適当ではない。

現段階において、ソフトウェアオブジェクト間の制約・関係を形式的かつ正確に記述したソフトウェアデータベースは存在しない。

1.2 研究概要

1.2.1 目的

本研究のゴールは、ソフトウェアデータベースの構築方法の確立である。

本研究は、材料として UNIXコマンド、道具として PCTE を用い、ソフトウェアデータベースの構築実験を行なう。ソフトウェアデータベースには、形式的でないオブジェクトの処理が不可欠だが、本研究はその問題には触れず、ソフトウェアオブジェクト間の制約・関係や動的情報を形式的に記述することを重視する。

最終的に実現した、制約を分類し、ソフトウェアデータベースの構築方法に有効な性質を抽出することを目指す。

1.2.2 アプローチ

本研究では以下の手順で、UNIXコマンドデータベースを作成し、ソフトウェアデータベースの構築実験を行なう。

1. 対象となる UNIX コマンドを選択する。
2. UNIX コマンドから関係を抽出し、実体・関係・属性 で記述した ERA 図で記述する。
3. UNIX コマンドから性質を抽出し、それをプログラムとして実現する。

UNIX コマンドデータベースを作成した後は、プログラムとして実現した、もしくは実現しなかった制約を分類する。この際、主に難しさによる分類を行ないソフトウェアデータベースの構築方法の手がかりとなる性質を抽出する。

1.2.3 成果

ソフトウェアオブジェクト間の制約・関係には記述が難しくその完全な実現が不可能なものが多数存在する。本研究では、そういう制約に対していくつかの支援方法を示し、それを実際来实现していく過程で、ソフトウェアの制約の記述方法は分類できることが確認できた。

現在、制約の分類は大きく分けて、位置情報に関する制約、意図に関する制約、動的情報に関する制約の3つが確認できている。

1.3 論文構成

本論文は7章からなり、その構成は以下の通りである

第1章 本章。

第2章 本研究で材料として使用した UNIX コマンドの特徴を述べた後で、既存の方法では UNIX のファイルシステムにおける制約・関係の情報を得るには不十分であることを述べる。また、問題を具体化する為に UNIX コマンドにおける、ソフトウェアデータベースが必要となる例題をいくつか挙げる。

第3章 本研究で道具として使用した Emeraude PCTE についての概要を述べ、PCTE の仕様に基づき SDS や、ツールについて述べる。

第4章 まず、本研究でソフトウェアオブジェクト間の関連を記述する方法として用いた階層構造を提案し、クラス図が静的な関係、インスタンスが動的な関係を記述・参照するのに、メタ・クラス図がそれらを形式的に記述するのに必要であることを説明する。次に、種々の UNIX コマンドにおけるソフトウェアオブジェクト間の関係を抽出し、ERA 図で記述する為に UNIX のファイルシステムにおける実体型、関連型の定義を行なう。その後で UNIX コマンド `cat`, `man`, `tar` におけるオブジェクト間の関係を ERA 図を用いて記述する。また、動的情報の ERA 図を記述する為に インスタンス図の実現可能性について述べた後で、その部分実現方法を提案する。

第5章 本研究で実現した `ta` のオプションに関する制約と、環境変数 `PAGER` に関する制約について述べる。それぞれの制約を実現する機能について述べた後で、その実現方法を示す。まず、実現方針を述べた後でそのソースコードのキーとなる部分を抜粋し、用いた PCTE ライブラリの関数の機能を説明する。最後には実行結果を示し、作成した UNIX コマンドデータベースの評価を行なう。

第6章 第4章、第5章を経て作成した UNIX コマンドデータベースを評価する。第5章では本研究で作成した UNIX コマンドデータベースそのものの評価を行なったが、第6章では 本研究で用いた手法で UNIX コマンドデータベースを作成することに対しての評価を行なう。次に、制約の分類を行ない、分類結果を示した後で、その分類が一般的なソフトウェアデータベースに適用できるかを C のプログラムのデータベースを作成することを想定に当てはめ、ここで得た分類が UNIX コマンドに限定されたものではないことを述べる。最後に本研究で用いたメタ情報が C のプログラムのデータベースを例に必要なものであることを示す。

第7章 本研究をまとめ、結論として ソフトウェアデータベースにおける制約・関係の記述は難しい。形式的な記述と非形式的な記述の両方が必要。メタ情報が必要。制約は分類可能であることを述べ、制約の分類が可能であることから、本研究をすすめていくことはソフトウェアデータベースの諸問題を解く手段として有効であることを述べる。最後に本研究を進めていく上での今後の課題として、さらに多数の UNIX コマンドの制約や関連を着手すること、制約の分類により細分化したソフトウェアデータベースの問題の考察、インスタンス図の半自動作成ツールの作成があることを述べる。

第 2 章

UNIX コマンドの概要

- UNIX のコマンドは相互に複雑に関係し合っている。
- その複雑な関係をユーザに伝える手段は現在のところ存在しない。

2.1 UNIX のシステム

2.1.1 シェル

シェルとはユーザが UNIX コマンドを使用する際の入力を解釈実行するプログラムである。シェルには以下の役割がある。

- ユーザインタフェース
ユーザの要求をコンピュータで実現する為の、ユーザとコンピュータの意思疎通の役割をもつ。
- 環境設定の道具
エイリアス機能や環境変数、設定ファイルの設置などにより、個々の環境をカスタマイズすることができる。
- プログラミング言語
シェルの内部コマンドやシェルが呼び出すことができる UNIX コマンドを利用して、プログラミングを行なうことができる。`.cshrc` はプログラミング言語としてのシェルを利用したプログラムの一例。

パイプとリダイレクション

シェルはコマンドの入出力を切り替える機能を持ち、複数のコマンドをつなぎ合わせるパイプや、コマンドとファイルをつなぎ合わせるリダイレクションがある。

これらの機能を利用するには コマンドと入出力との関係を適切に理解する必要がある。

UNIX コマンドはコマンドがコマンドを呼び出すことがあるので、相互に関係しているが、シェルのプログラミング言語としての機能や パイプ、リダイレクションを使うことにより、さらに複雑に関係してくる。

2.1.2 コマンド

UNIXコマンドには以下の特徴がある。

- コマンド名から動作を推測することが困難。
- インターフェースが統一されていない。
例えば、複数のコマンド間で使われている同じ名前のオプションが、同じ内容を示すとは限らない。
- 環境変数の設定が必要。
UNIXコマンドには環境変数を呼び出すものがあり、この環境変数の設定によりコマンドの実行結果は異なる。

この為、ユーザは多数ある UNIXコマンドの使用方法を覚えるのは困難で、必要なときにコマンドの使用方法を調べることが必要である。

2.2 実験材料としての UNIX コマンド

UNIXコマンドは身近であるため、その理解しやすさ故に実験材料として扱いやすい。また、UNIXのファイルシステムはさまざまなオブジェクト間で制約・関係が成り立っており、引数や環境変数、ファイルデータなど、実験毎に異なる情報も多いが、一般のソフトウェアに比べて比較的簡潔な形になっている。これは UNIXのファイルシステムが制約・関係や動的情報といった、ソフトウェアデータベースの問題の一部を含みつつ、簡単な形になっていることを意味している。また UNIXコマンドはソースが公開されているので、それを用いてソフトウェアの持つ意味を読みとることができる。

これらの理由により、UNIX コマンドはソフトウェアデータベース構築の実験材料として適切なものであると判断した。

2.3 情報参照の現状と問題点

現在、UNIXの情報入手する為の利用手段として以下のものが存在する。

- マニュアル
- ドキュメント
- 書籍
- Web
- 他のユーザ
- ソースコード

マニュアル

マニュアルはUNIXコマンド `man` などを利用することで、参照することができる。各々のコマンドに対して以下の情報が記述されている。

- コマンドの使用目的
- コマンドの引数
- コマンドの機能（オプションの情報など）
- 関連するコマンド
- 関連する環境変数
- 関連するファイル
- 使用例

マニュアルはユーザがコマンドの使用方法を参照することを目的に記述したものである。したがってソフトウェアオブジェクト間の制約・関連や動的情報は十分に記述されていない。また、記述が非形式的である為、検索方法が、キーワード検索に限られる。

ドキュメント

ドキュメント にはテキスト形式で記述したものと、ハイパーテキスト形式で記述したもの、また java のプログラムに限定した javadoc がある。

- テキスト 形式

テキスト形式で書かれたドキュメントには、ユーザがコマンドの使用方法を参照することを目的に記述したものや、HOWTO のようにユーザが疑問を持ちやすい箇所の解決方法を記述したものがある。いずれも 記述が非形式である為、検索方法が grep などのキーワード 検索に限られ、また動的情報も不十分である。

- ハイパーテキスト 形式

ハイパーテキストは、画像の貼りつけや リンクなどの機能があるが、この機能を十分に生かしたドキュメントは現在のところ存在しない。リンクなどの機能をつかえば、ソフトウェアオブジェクト間の関係などが形式的に記述できることが期待できるが、現段階では自然言語での非形式的な記述に頼っている。

- javadoc

ソースにマニュアルの情報を記述することにより、ハイパーテキスト形式のドキュメントを自動的に作成する。ソースは開発者が記述するものなので、正確で、開発者の意図に沿った情報を得られることが期待できる。現段階では自然言語での非形式的な記述に頼っている。

書籍

初心者に馴染みやすい、ユーザがチェックやメモを入れやすい、などの利点があるが、計算機による支援が行なわれない。また、費用がかかる、更新が不便等の問題点がある。

Web

Web はネットワーク上に存在する情報で、情報量が多いという利点があるが、その反面、情報が散在していて目的の情報を得にくいという問題点がある。一般的にハイパーテキスト形式で情報が記述されているが、ドキュメント同様、その機能を十分に生かしきれておらず、ソフトウェアオブジェクト間の制約・関係を形式的かつ正確に記述したものは存在しない。

他のユーザ

動的情報を得ることができるが、ユーザが常に使える環境ではない、目的の情報を持っているとは限らないという問題点がある。

ソースコード

ソースコードはプログラムであり、コマンドそのものである。したがって十分な情報量を持ち、また形式的な記述を持つ。しかし、情報量が膨大であり、ユーザが読みやすい形になっていないという問題点がある。

2.4 例題

まず、問題を具体化するために、UNIX コマンドにおけるいくつかの例題とその実現可能性を検討する。

例題 1 コマンド `tar` のオプション `x` と同時に使えないオプションは何か。

オプション `c`, `r`, `t`, `u` は `x` と同時には使えない。

この実現は容易である。解答はインストール前に静的に一意に決まり、かつ形式的に表現できるからである。

例題 2 環境変数 `MANPATH` にセットすべきパス名の候補は何か？

コマンド `man` では参照するマニュアルファイルの場所を指定する為の環境変数 `MANPATH` の設定が必要である。この設定が不適切であると、マニュアルファイルが存在しない、もしくは期待するマニュアルでないものが表示されるといった事態に陥る。

この例題の実現はやや難しい。解答はインストール後に一意に決まり、かつ形式的に表現できる。以下は部分的な実現方法であるが、いずれも問題がある。

- ほとんどの環境で共通である、`/usr/man/` や `/usr/local/man/` などが組み込まれているかチェックする。
 - － 個々の環境に対応していないので不十分
- `.cshrc` などで、他のユーザの `MANPATH` の設定を参照する。
 - － `.cshrc` はプログラムであるため、どの設定が個人の環境に合った設定か、プログラムで判断するのは困難。

- `find` コマンドで、マニュアルファイルのある場所を探す。
 - 見つけたディレクトリが、個人の環境に合ったマニュアルのファイルが置いてあるのか判断することは困難。
- 管理者が新しいマニュアルを設置したとき、特定の場所に位置情報を記述する。
 - マニュアルだけなら現実的な方法であるが、それ以外の いろいろな 設定に対してもこの方法を用いると、管理者の仕事は膨大なものとなる。

例題 3 環境変数 `PAGER` にセットすべきコマンドは何か？

コマンド `man` では、取得したマニュアルファイルは環境変数 `PAGER` で設定しているコマンドにより処理する。したがって、この設定が不適切であると適切なマニュアルファイルを得ることができても、ユーザに読める形にはならない。例えば、`PAGER` に コマンド `ls` をセットしても システム上では何の問題もないが、表示されるのはマニュアルの中身ではないので適切な動作とはいえない。

あるコマンドが `PAGER` にセットしてよいコマンドであるかどうか判断することは困難である。なぜなら、`PAGER` の「長いテキストをユーザにとって読みやすく少しずつ表示する」という意図はプログラミングが難しいためである。

以下に実現案を示すが、いずれも問題がある。

- 特定の場所に意図に沿ったコマンドを列挙して記述しておく。
 - 新しく入手したり、開発したソフトウェアには対応できない。
- ソフトウェアデータベースに記述されている関係を参照し判断する。例えば、`PAGER` の場合は標準入力から読み取ったデータを標準出力しているかを判断する。
 - `mule` や `netscape` 等は、`PAGER` に設定しても マニュアルを参照できるが、標準出力に出力していないため、この方法では不十分。
- コマンドにプログラムの意図を示す属性 `type` を与える。例えば `type` には `pager`, `manual`, `compiler`, `printer`, `browser` などがあるとする。
 - `netscape` や `mule` にも対応させるには、`type` が `pager` でないものも考える必要がある。
 - この方法でも完全実現は不可能であるが、開発者の手間は意図を型として記述するだけでよく、さらに 比較的 正確に近い判断が出来るので、有効かつ現実的な手段であると言える。

例題 4 コマンド `man` を使ったとき、どこのパスを検索したか。

コマンド `man` の検索するパスは動的に決定する。オプション `M` でパスが指定してあれば、指定したパスのみを検索するが、そうでなければ、環境変数 `MANPATH` にセットされているパスを前から順番に検索する。

これは `man` コマンドの検索の動きをプログラムで再現することにより、実現できる。しかし、コマンドの動きとはコマンドのプログラムそのものである為、現実的な方法ではない。

例題 5 `nemacs` 用の `.emacs` を `mule` 用の `.emacs` に変更したい。

これは非常に難しい。この問題は本質的に移植の自動化が必要だからである。変更された関数のリストがあれば、変更が必要な箇所を示すことはできる。

例題 6 UNIX コマンドの中で、パスを設定する環境変数を知りたい。

ソフトウェアオブジェクト間の関係を形式的かつ統一的に記述することによって実現できる。パスを設定する環境変数は“環境変数”と“ファイル”が“パスを決定する”という関係が成り立っている。これを UNIX コマンド全体の中から検索可能にするには、UNIX コマンドデータベース全体を統一的に記述する為のメタ情報が必要である。

第 3 章

PCTE

3.1 PCTE とは？

PCTE とは Portable Common Tool environment の略で、さまざまなソフトウェア・ツールを統合的に利用する環境の一つである。

PCTE の仕様は ERA モデル (実体・関連・属性を記述したモデル) にリンク属性など、特定の性質を付加した SDS (Schema Definition Set) により、記述し、また意味的制約もサポートしている。

PCTE プロジェクト は、1983 年から欧州共同体による ESPRIT 計画の一環としてスタートし、1983 年、PCTE1.4 の仕様が公開された。一方、セキュリティ機能などを更に今日かし、NATO の要求を満たすような仕様を目指しているものに PCTE+ がある。PCTE+ は NATO の IEPGTA-13¹ という作業グループで作成された。更に 1988 年、NATO 案は ECMA² に提案され、TC33³ というタスクグループが発足し、PCTE+ を基にした参照モデルの検討を開始し、1990 年に ECMA 標準 149 の一般仕様が公開された。
[5]

PCTE の標準化

PCTE は ESPRIT プロジェクト、NATO 拡張、ECMA 標準、ISO へと発展し、標準化が続けられてきている。現在、既に ISO 13719-(1PCTE 概要)、ISO 13719-(2C プログラミング言語バインディング)、13719-(3Ada プログラミング言語バインディング) の

¹Independent European Programme Group, Technical Area 13

²European Computer Manufacturers Association

³Technical Committee 33

国際標準化が完了している。

3.2 Emeraude PCTE

本研究では PCTE を実現するソフトウェアの一つ、Emeraude PCTE を用いて UNIX コマンドデータベースの作成を試みる。Emeraude PCTE は `esh` (図 3.1) と呼ばれる Emeraude PCTE 特有のシェルにログインすることによって、オブジェクトの参照、生成を行なう。C 言語やシェルスクリプトを用いて制約をプログラミングでき、プログラミングしたものをツールと呼ぶ。

Emeraude V12 は PCTE 1.5 に基づいている。PCTE 1.5 は 1988 年に仕様が完成したもので、1990 年に完成した ECMA PCTE より古いものである。

```
esh$
esh$ els
1..
ArtificialIntelligence.lec
DiscreateMathematics.lec
ProgramingMethology.lec
biology.lec
esh$
esh$ obj_create lecture ComputerSystem.lec
esh$
esh$ els
1..
ArtificialIntelligence.lec
ComputerSystem.lec
DiscreateMathematics.lec
ProgramingMethology.lec
biology.lec
esh$
esh$ link_delete biology.lec
esh$
1..
ArtificialIntelligence.lec
ComputerSystem.lec
DiscreateMathematics.lec
ProgramingMethology.lec
esh$
```

図 3.1 `esh` の実行例

3.3 実験道具としての PCTE

PCTE は ERA モデルを拡張した SDS を用いて記述するため、関係を用いてオブジェクトを参照するのに優れている。また、SDS は多重度や存在関係等の制約の記述をサポートしており、サポートしていない制約は C 言語を用いてプログラミングすることにより、ツールとして実現できる。本研究は 制約・関係の記述を重視しているため、以上の利点から PCTE は使用するソフトウェアとして適当なものであると判断した。

また、UNIX コマンドデータベース全体の中から検索を行なうにはそれらの情報を統一的に記述する、メタ情報が必要である。PCTE では UNIX コマンドデータベース用のスキーマを一つに限定することによって、UNIX コマンドデータベース全体を統一的に記述するメタ情報とすることができる。

3.4 SDS 図

SDS とは ERA モデルに特定の性質を付加したものである。

3.4.1 ERA モデル

ERA モデルとは Entity (実体) と Relation (関係) と Attribute (属性) を示したモデルであり、実体と実体の間の関係を理解しやすい。

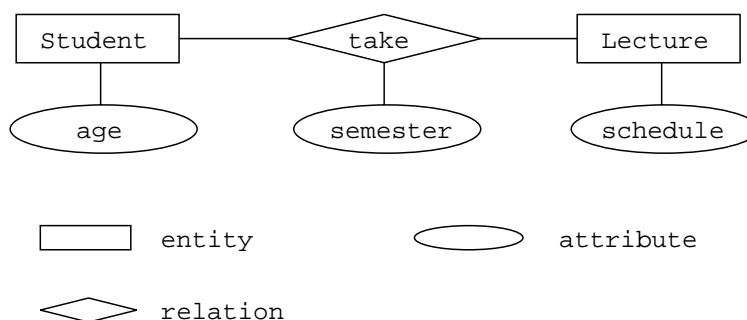


図 3.2: 学生と講義の関係を示した ERA 図

図 3.2 は学生と講義の関係を示した ERA 図である。これは、学生と講義の間に `take` という関係が成り立っていることを意味しており、長方形は実体、菱形は関係、楕円形は属性を表している。

しかし、学生と講義の間には「同じ時間割の講義はとることができない。」等の制約があるが、この制約は ERA モデルで表わすことができない。

3.4.2 リンクカテゴリ

SDS 図には、リンクにカテゴリがある。カテゴリは、リンクの意味的な分類を行なう。ECMA 仕様ではリンクには次のプロパティがある。

- 始点依存性：リンクの始点のオブジェクトに、このリンクの生成、削除を行なう権限が必要。
- 参照完全性：リンクの終点のオブジェクトを削除できない。逆リンクも参照完全性をもつ。
- 存在特性：リンクの終点のオブジェクトを生成できる。リンクを削除すると、リンク終点のオブジェクトも削除される。
- 構成特性：リンクの終点のオブジェクトが始点のオブジェクトの構成要素となる。

リンクのカテゴリはリンクのプロパティの組合せにより定まる。

カテゴリ	始点依存性	参照完全性	存在特性	構成特性
COMPOSITION				
EXISTENCE				
REFERENCE				
IMPLICIT				
DESIGNATION				

図 3.3: リンクのカテゴリとプロパティ

Emeraude PCIE でのリンクカテゴリは、composition,reference,implicit,system implicit の 4 つである。

- Composition Link：このカテゴリに属するリンクは始点オブジェクトが終点オブジェクトを生成する能力をもつ。Emeraude PCIE ではオブジェクト `common_root` 以外のオブジェクトは全て Composition link の副産物として存在している。Composition link が削除されれば、終点のオブジェクトは消える。

- **Reference Link** : このカテゴリに属するリンクはリンクの両端のオブジェクトが純粋に意味的な関係をもつ。
- **System implicit Link** : 逆リンクにのみ用いられる。システムが自動的に付加するもので、ユーザが SDS を作成する際にこのカテゴリのリンクを指定することはない。
- **Implicit Link** : 逆リンクのみに用いられる。System implicit Linkのようにシステムが自動的に付加するものではなく、明示的な局所名をつけて参照したい場合に用いる。

Emeraude PCTE の `esh` で、オブジェクトを削除するコマンドはなく、図 3.1 のように、リンクを削除するコマンド `linkdelete` で `Composition link` を削除することによってオブジェクトを削除する。[5]

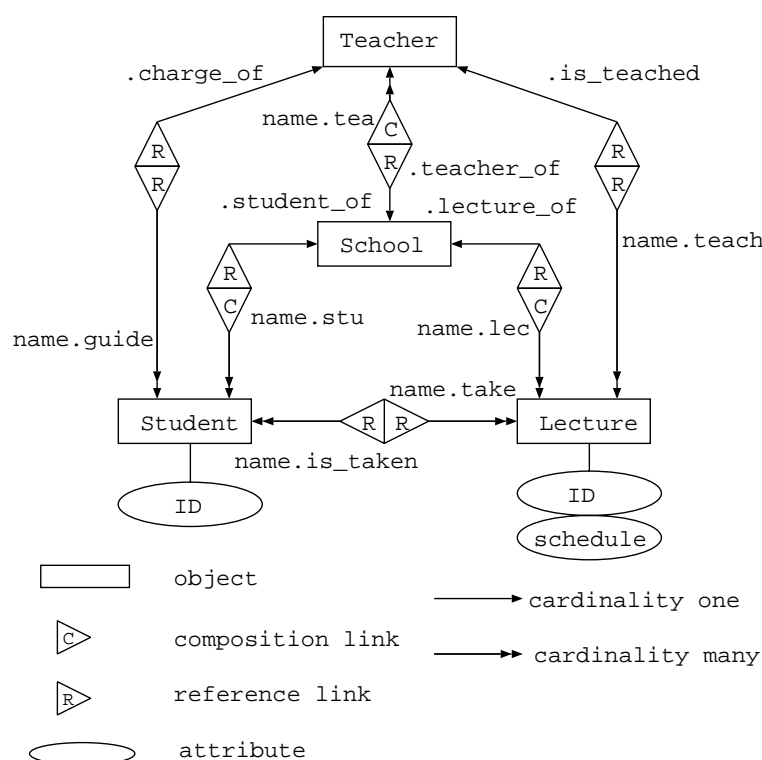


図 3. 4: 学校内の関係を示した SDS 図

図 3. 4は学校内の関係を示した SDS 図であり、図中には `Composition link` と `Reference Link` が見られる。この例の場合、新しい `Student` や `Teacher` や `Lecture` のオブジェクト

を作ろうとした場合、School から CompositionLink を用いることによって生成される。また、ReferenceLink は対象のオブジェクトを参照できることを意味している。

以下、オブジェクトとその関連を示す図は、繁雑さを防ぐ意味で SDS 図ではなく、ERA 図を用いる。前で述べたように、SDS 図は ERA 図に特定の性質を付加したものなので、ERA 図で表現できるものは、SDS 図でも表現できる。

PCTE 上で記述する際は SDS 図で記述する。

3.5 ツール

3.5.1 SDS で実現する制約と ツールにより実現する制約

図 3.4 の関係でも、「学生は同じ時間割の講義は受講することができない。」があるとする。この制約は SDS 図でも表現することは難しい。

SDS 図で表現しにくい制約は Emeraude PCIE ではプログラミングを行なうことにより、ツールとして実現する。

- 実体に対する制約

- インスタンス数に関する制約
- 属性値間の制約

これらは SDS で表現不可能である為、ツールを用いて実現する。

- 関連に対する制約

- インスタンス数に関する制約 … SDS で表現不可能である為、ツールを用いて実現する。
- 属性値間の制約 … SDS で表現不可能である為、ツールを用いて実現する。
- 始点・終点をなす実体間の制約

- * 始点と終点の属性値間の制約

SDS で表現不可能である為、ツールを用いて実現する。

- * 始点・終点の包含関係 (包含・一致・分離・非分離⁴)

包含・一致に関してはオブジェクトの継承関係を用いて SDS で表現はでき

⁴包含: $A \subset B$, 一致: $A = B$ 分離 $A \cap B = \phi$, 非分離: $A \cap B \neq \phi$

るが、実際には継承関係を用いても包含関係などに捕らわれず、オブジェクトを生成できるので、実現しているとはいえない。また、分離・非分離は SDS で表現できない為、いずれも、ツールを用いて実現する。

* 始点・終点の存在依存性

SDS ではリンクに存在特性を付与することにより、表現できる。Emeraude PCTE では存在特性は `CompositionLink` のみが持つ。⁵

* 始点・終点の構成関係

SDS ではリンクに構成特性を付与することにより、表現できる。構成特性は `CompositionLink` のみが持つ。

- 実体と実体の間の制約

実体間の制約は関連の制約として記述する。

- 関連と関連の間の制約

- 関連間の存在数の制約
- 関連属性値と関連の存在の制約
- 関連属性値の制約

これらはいずれも、SDS で表現できない為、ツールを用いて実現する。

- 実体と関連の間の制約

- 実体に対する関連の存在・数に対する制約
カージナリティ範囲を用いて表現可能。Emeraude PCTE では、数の指定はできず、多重度を用いて 1 個だけか、それ以上も可能かの指定になる。
- 実体属性値と関連の存在・数に対する制約 … SDS で表現できないため、ツールを用いて表現する。
- 実体属性値と関連属性値の間の制約 … SDS で表現できないため、ツールを用いて表現する。

[7]

⁵ECMA 仕様では `Existence Link` も持つ。

3.5.2 ツール実現の為にライブラリ関数

Emeraude PCTE では、PCIE の C 言語用のライブラリが用意されており、esh で扱う、オブジェクトやリンク、属性の操作が一通りできるようになっている。

機能	関数
オブジェクト生成	<code>crobj(char *origin, char *l, char *type, int mode)</code>
リンク生成	<code>crlink(char *origin, char *l, char *dest)</code> <code>crlinkr(char *origin, char *l, char *dest, char *rev)</code>
オブジェクト・リンク削除	<code>dllink(char *origin, char *l)</code>
属性の値を参照	<code>getattr(char *path, char *attr, int size, attr value)</code> <code>getlattr(char *path, char *l, char *attr, int size, attr value)</code>

図 3.5: PCIE ライブラリの関数の例

図 3.5 で記述している関数の他には特定のオブジェクトからリンクを抽出する `lslinks`、属性値を記述する `setattr` など、多数存在する。

例として、図 3.4 で「同じ時間割の講義を受講することはできない」という制約を考える。これは、特定の学生がある講義を受講する `take` という関係を結ぼうとしたとき、(1) 最初に受講しようとする講義の属性 `schedule` を参照し、(2) 学生が既に `take` の関係にある講義を列挙し、(3) それらの講義の属性 `schedule` を参照し、(4) 受講しようとしている講義の `schedule` と同じものがなければ、`take` の関係を結ぶことで実現できる。それぞれの過程で使用する PCIE ライブラリの関数は (1) `getattr`, (2) `lslinks`, `getattr`, (4) `crlinkr` である。(2) では、`lslinks` は引数としてオブジェクトの `definitionId` が必要なので、それを得るためにオブジェクトの状態を `objstat` 構造体に格納する関数 `getobjstat` が必要となり、また `lslinks` で得るのはリンクの状態を格納した `linkstat` 構造体である為、それから 講義名を得るための `linkname` 関数が必要である。

第 4 章

UNIX コマンド の関連の記述

4.1 階層構造

本研究では UNIX コマンドデータベースを作成する際、以下の 3 つの階層に分かれるようにした。

- メタ・クラス図 … UNIXコマンド 全ての実体型、関連型の定義を示した図
- クラス図 … 特定の UNIXコマンド の実体間の関係を示した図
- インスタンス図 … 具体的な実行例に対し、具体的なオブジェクト間の関係を示した図

これを階層構造と呼ぶ。[1] 引数とファイルの関係など、静的な関係図を参照するときには、クラス図を用いる。また、具体的なファイル名とパラメータの値など、動的な情報を参照するときにはインスタンス図を用いる。関係し合っている為、複数のコマンドの関係図を統一的に記述することが必要である。この為、複数の UNIX のコマンドの共通の概念となる関係図、メタ・クラス図を用いた。

4.1.1 メタ・クラス図

メタ・クラス図は UNIXコマンドデータベースの全ての実体型と関連型を定義する為のもので、1 つだけ作成する。Emeraude PCTE では SDS 図を作成し、メタ・クラス図を表記する。

メタ・クラス図を導入することにより、複数のコマンドのクラス図の記述が統一的となり、比較が行ないやすくなる。

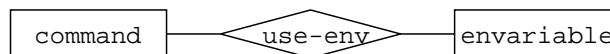


図 4.1: メタ・クラス図の例

4.1.2 クラス図

クラス図は UNIX コマンドの静的な関連を記述するもので、UNIX コマンド 毎に作成する。メタ・クラス図で定義された実体型と関連型を用いて作成する。Emeraude PCTE 上では作成した メタ・クラス図をワーキングスキーマに与え、e sh 上でオブジェクトやリンクを作成することによって実現する。

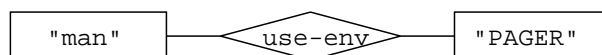


図 4.2: クラス図の例

4.1.3 インスタンス図

インスタンス図は具体的な実行例を与えたときに記述することができる、具体的なオブジェクト間の関係を示した図である。クラス図で定義した実体間の関係を用いて作成する。

インスタンス図を導入することにより、実行時の具体的なオブジェクトを参照し、意図しない実行結果の究明に役立てることができる。

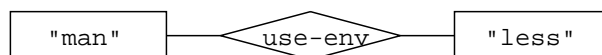


図 4.3: インスタンス図の例

4.2 メタ・クラス図の作成

UNIX コマンドデータベースの作成にあたり、まず最初に行なうことはメタ・クラス図の作成である。メタ・クラス図の作成は実体型、関連型、属性型の定義を意味する。

まず実体型の定義を行なう。

実体型	属性	説明
command	name type	UNIX コマンド
parameter	name	引数
environment	name, default	環境変数
file	name	ファイル
data	name type	内部データ
socket	name type	ネットワーク上に存在するプリンタや端末など
command	name type	コマンドが使用するコマンド

図 4.4 : 実体型の定義

- **command** : ユーザが入力、使用するコマンドを表す実体型を **command** 型とする。**command** 型には以下の属性があるとする。
 - **name** 使用するコマンドの名前を示す。
 - **type** : コマンドの意図を示す属性。**page edit browse print** などがあるとする。
- **parameter** : ユーザが入力するコマンドの引数を表す実体型を **parameter** 型とする。**parameter** 型には以下の属性があるとする。
 - **name** コマンドには複数の引数をとるものもあるので、属性 **name** を用いて他の引数と区別する。
- **environment** : コマンドが使用する環境変数を表す実体型を **environment** 型とする。**environment** 型には以下の属性があるとする。
 - **name** 環境変数名を示す。
 - **default** 環境変数に値をセットしていない場合に用いられる値を示す。
- **file** : コマンドを使用することにより扱うファイルを表す実体型を **file** 型とする。また、標準入力、標準出力、標準エラー出力も **file** 型として扱う。**file** 型には以下の属性があるとする。

- `name` : ファイルの名前を示す。
- `data` : コマンドを使用することにより扱うデータを表す実体型を `data` 型とする。
`data` 型には以下の属性があるとする。
 - `name` : 複数のデータを扱う際、属性 `name` を用いて他のデータと区別する。
 - `type` : データの型を示す。`text`, `postscript`, `gif`, `tar` などがあるとする。
- `socket` : ネットワーク上に存在する端末やプリンタなどを表す実体型を `socket` 型とする。`socket` 型には以下の属性があるとする。
 - `name` : 複数のソケットを扱う際、属性 `name` を用いて他のソケットと区別する。
 - `type` : ソケットの型を示す。`printer`, `www`, `terminal`, `ftp` などがあるとする。
- `com` : コマンドが使用するコマンドの型を `com` 型とする。`command` 型はユーザが使用するコマンドそのものを示すのに対し、`com` 型は コマンド 中で使用するコマンドが扱うファイルや環境変数などを含めた全体を示す。
 - `name` : コマンドの中で扱うコマンドの名前を示す。
 - `type` : `command` 型の属性 `type` と同じように意図を示す。`pager`, `editor`, `browser`, `print` などがあるとする。

実体型を定義すれば、次に関連型を定義すれば UNIX コマンドデータベースのメタ・クラス図が作成できる。メタ・クラス図を図 4.5 に示す。

このメタ・クラス図は ERA 図で、長方形が実体型、菱形が関連型、楕円形が属性を表している。

以上の実体型、関連型、属性型はこれから紹介する UNIX コマンド、`cat`, `man`, `tar`, `lpr` を元に定義したもので、UNIX コマンドデータベースの作成を続けていく上で、他の UNIX コマンドではこのメタ・クラス図では不十分、もしくは適当でない可能性がある。しかし、今後さまざまなコマンドのデータベースを作成していく際に、必要に応じてメタ・クラス図の修正を行なっていくことで、完全なメタ・クラス図に近付けていくことができる。

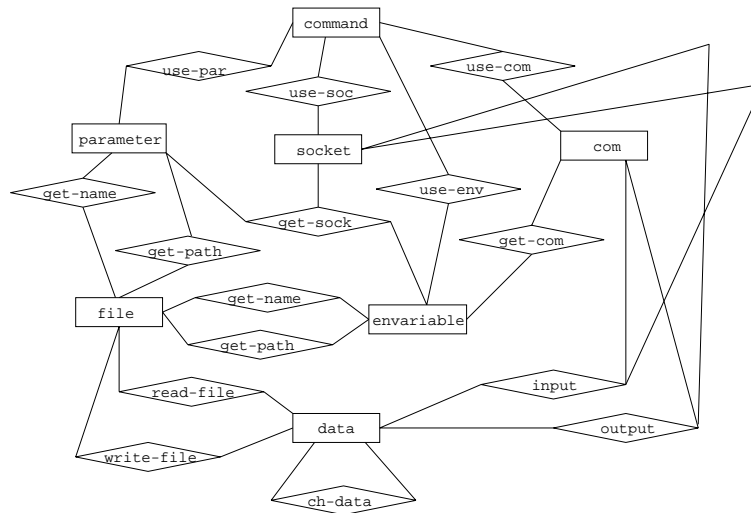


図 4.5: UNIX コマンドデータベースの メタ・クラス図

4.3 クラス図の作成

4.3.1 例

cat, man, tar, lpr のクラス図を以下に示す。

cat

cat コマンドの ERA 図を 図 4.6 に示す。

しかし、図 4.6 では以下の制約は表現できていない。

- ファイルのデータはテキスト形式でなければならない。
- 環境変数 LANG にはターミナルがサポートしている言語のコードをセットする。

man

man のコマンドの ERA 図を 図 4.7 に示す。

しかし、図 4.7 では以下の制約は表現できていない。

- 環境変数 MANP ATH はマニュアルのファイルが格納されているディレクトリのパス名をセットする。

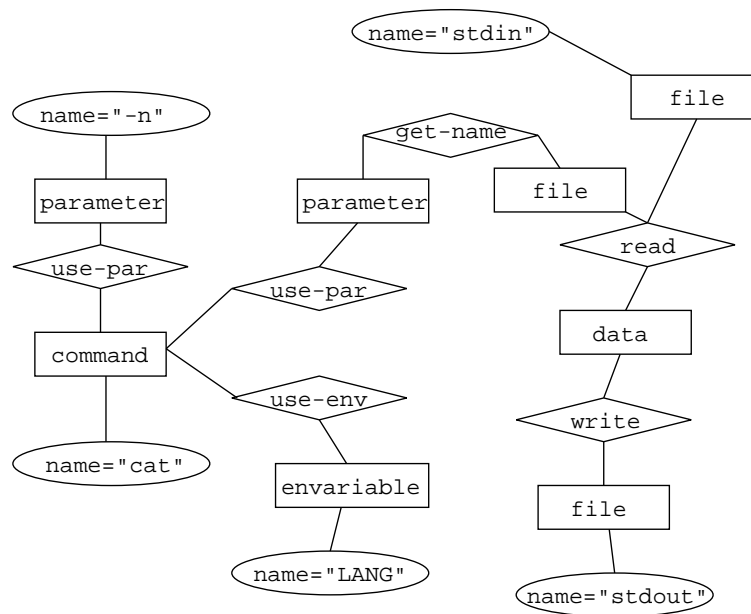


図 4.6: cat のクラス図

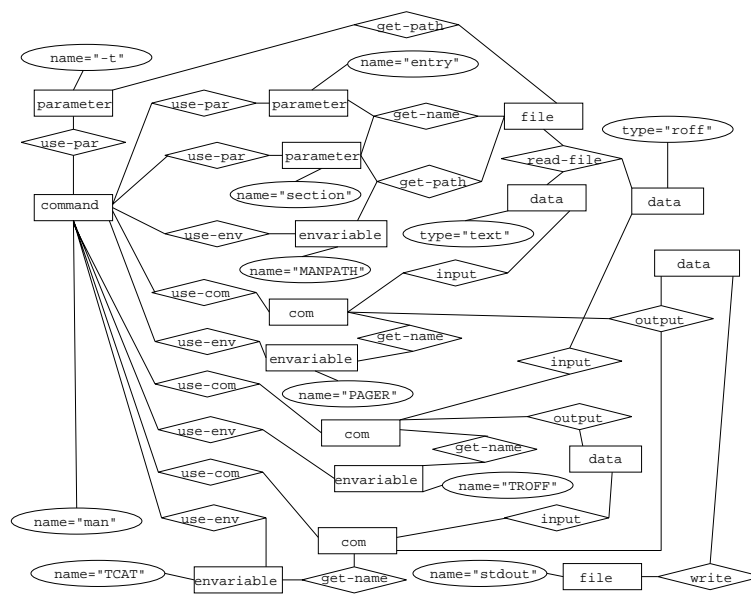


図 4.7: man のクラス図

- 環境変数 `PAGER` にはテキストファイルの中身を表示するコマンド名をセットする。
- 環境変数 `TROFF` は `roff` 形式のデータを処理する UNIX コマンド名をセットする。
- 環境変数 `TCAT` には `roff` 形式のデータを環境変数 `TROFF` の値で示されているコマンドで処理されたデータを適切な形でユーザに表示するコマンドをセットする。
(`TROFF` の値が `nr` なら、`less`、`more` など、`TROFF` の値が `ps` なら `lp` など)
- 引数 `entry` には `MANPATH` の中に存在するマニュアルの名前が入る。

`tar`

`tar` のコマンドの ERA 図を 図 4.8 に示す。

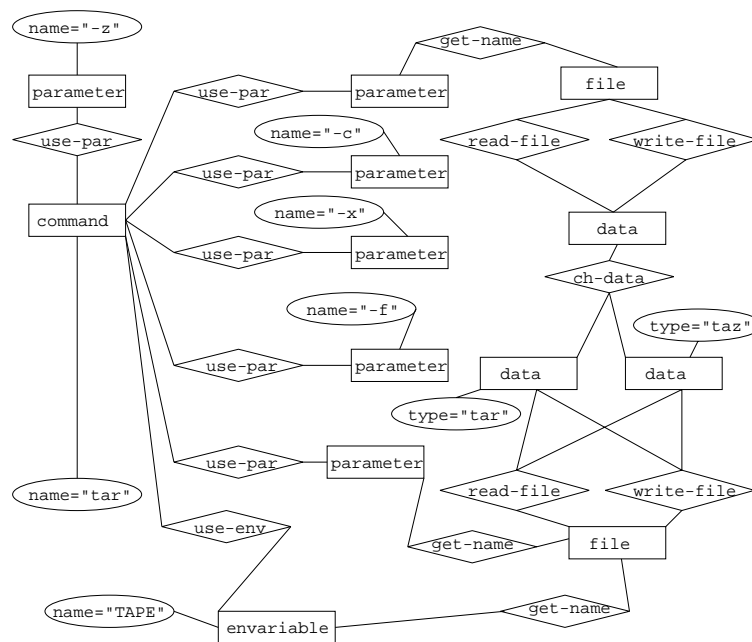


図 4.8 `tar` のクラス図

しかし、図 4.8 では以下の制約は表現できていない。

- オプション `x` と `c` は同時に使えない。
- オプション `z` がつければ、`type = "taz"` のデータを扱い、つかなければ `type = "tar"` のデータを扱う。

- オプション `x` がつけば、`type = "tar"` もしくは `type = "taz"` のデータを入力し、オプション `c` がつけば出力する。
- オプション `f` がついた時だけ `type = "tar"` もしくは `type = "taz"` のデータを示すファイル名を引数にとり、`f` がつかない時は環境変数 `TAPE` が呼び出される。

`lpr`

`lpr` のコマンドの ERA 図を 図 4.9 に示す。

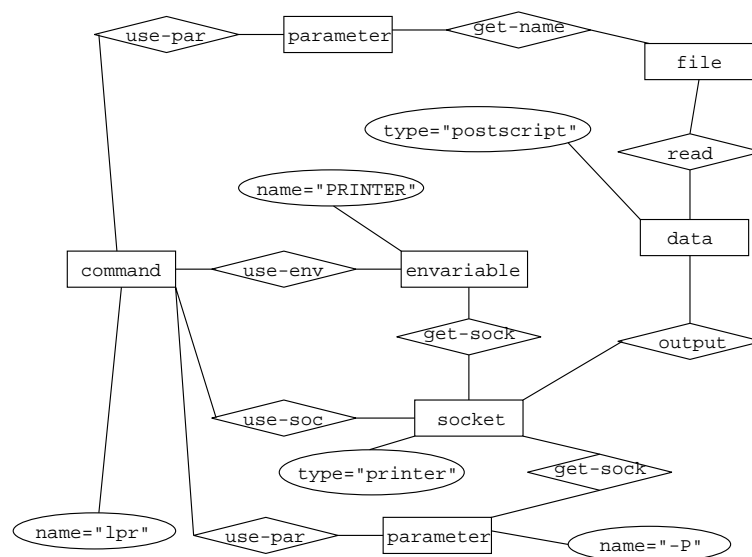


図 4.9: `lpr` のクラス図

しかし、図 4.9 では以下の制約は表現できていない。

- 環境変数 `PRINTER` 、オプション `-P` はネットワーク上に存在するプリンタの名前を指定する
- オプション `-P` でプリンタが指定されれば、環境変数で設定されているプリンタよりもそちらを優先する。

4.3.2 Emeraude PCTE 上での記述

Emeraude PCTE はオブジェクト `common_root` 以外のオブジェクトは `Composition Link` の副産物として存在する。よって新しいオブジェクトを生成しようとすると、既存の

オブジェクトから `CompositionLink` を生成することによって、オブジェクトを生成する。

Enraude PCTE のシステムには `dir` 型のオブジェクトが定義されている。これは UNIX のファイルシステムでいう、ディレクトリのような役割として使われる。`dir` 型のオブジェクトは `dir` 型から `CompositionLink` を生成することにより作ることができる。

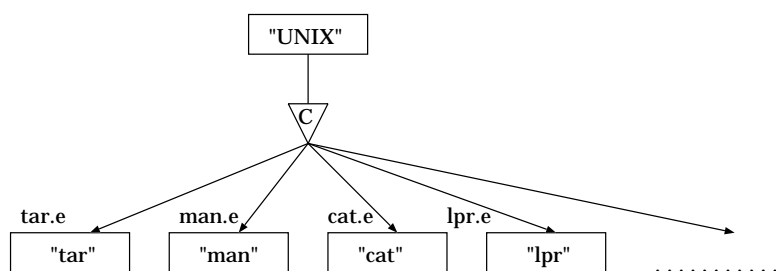


図 4.10: UNIX コマンドデータベースの構造 1

今回 UNIX コマンドデータベースを作成するに当たり、まず UNIX コマンドデータベースのあるディレクトリを示す、`dir` 型のオブジェクト `unix.e` を作成した。その下でさらに UNIX コマンドの名前を示す、`dir` 型のオブジェクト `[command_name].e` を作成した。(図 4.10)

さらに、`[command_name].e` のオブジェクトの下で、`command` 型、`parameter` 型、`environment` 型、`file` 型、`data` 型、`socket` 型、`com` 型のオブジェクトを生成し、それらの間で図 4.11 の関係が成り立っているスキーマ図を作成した。

図 4.11 で示しているのは全て `CompositionLink` であり、`dir` 型が UNIX コマンドデータベースに必要な全てのオブジェクトを生成することを意味してる。また、図 4.5 で示している関係は全て `ReferenceLink` である。

4.4 インスタンス図の作成

4.4.1 自動作成の必要性和実現可能性

全てのインスタンスをソフトウェアデータベースにあらかじめ記述することは不可能である。その為、全てのインスタンスに対応する為にはクラス図からインスタンス図の作成の自動化が必要である。しかし、インスタンス図を自動作成するツールを作成するには、コマンドのセマンティクスを詳細に知り、かつそれをプログラムとして表現する必要がある。

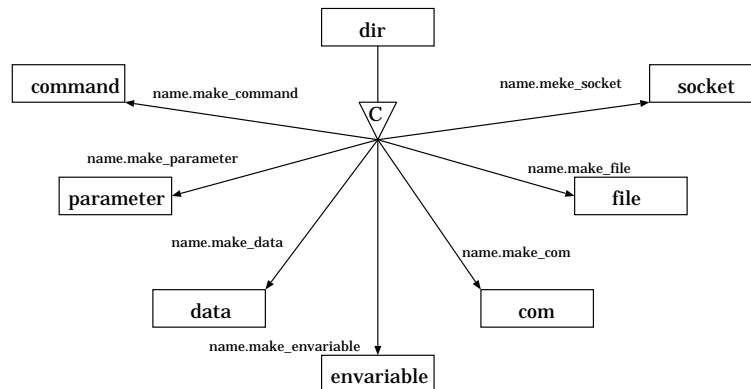


図 4.11: UNIX コマンドデータベースの構造 2

るため、現実的ではない。しかし、多数の UNIX コマンドで共通の動作は、インスタンス図の一部を生成するツールとして、ある程度は実現可能であり、かつ有効だと考える。

4.4.2 インスタンス図の自動作成

com 型の変換

UNIX では、複数のコマンドが相互に複雑に関係し合っている。その場合の多くはコマンドが内部のデータを、もう 1 つのコマンドが標準入力として入力するものである。

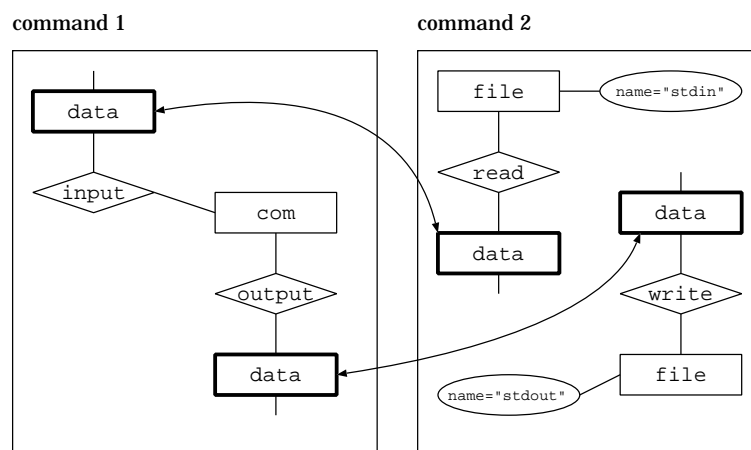


図 4.12: com 型の変換

その場合、図 4.12 で、command1 の内部で使用するコマンドが command2 である場

合、command1 と c o m m a n d 2 には図のような関係構造を持つことが期待できる。その場合、c o m m a n d 1 が呼び出すコマンドに入力するデータと c o m m a n d 2 が標準入力により得られるデータ、また c o m m a n d 1 が呼び出すコマンドから出力されたデータと c o m m a n d 2 が標準出力に出力するデータは同じものである。この場合その同じデータを 1 つのオブジェクトとすることにより、c o n 型を変換することができる。

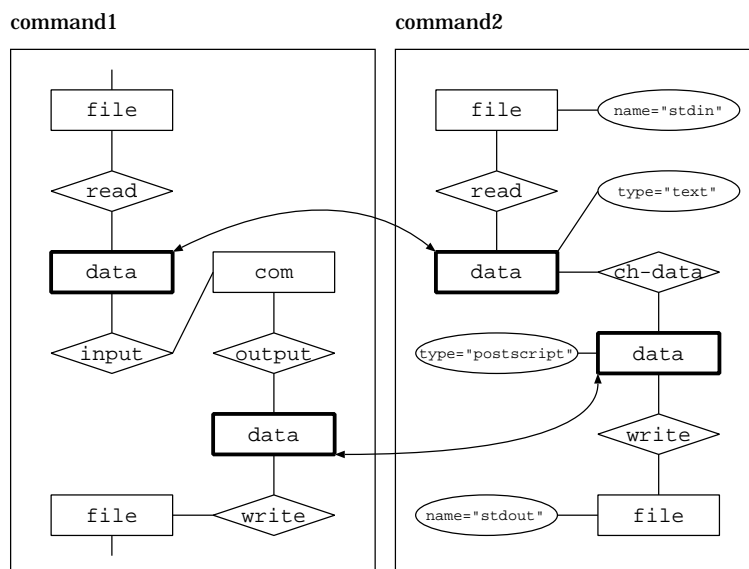


図 4.1 3: c o n 型の変換の例 (変換前)

例えば図 4. 1 の関係を持つ c o m m a n d 1 があり、c o m m a n d 1 が呼び出すコマンド c o m 型の実体が c o m m a n d 2 に決定したとすると、図 4. 1 のようになる。

c o n 型を変換することにより、クラス図まででは見ることができなかったコマンド内で呼び出すコマンドの内部の関係を記述することができ、具体的なオブジェクトを参照することができる。

パスの指定によるファイルの決定

コマンドのいくつかはコマンド内部で、必要なデータを得るためにその内部でファイルの検索を行なう。検索するファイル名はコマンドによって違うので、ファイル名を決定するツールを作成することは、UNIX コマンド 毎に作成しなければならず現実的でない。検索パスの決定も、例えば環境変数でのパス指定と、オプションでのパス指定などがあり、複数の方法で与えられているとき、どのように扱うかはコマンドに委ねられて

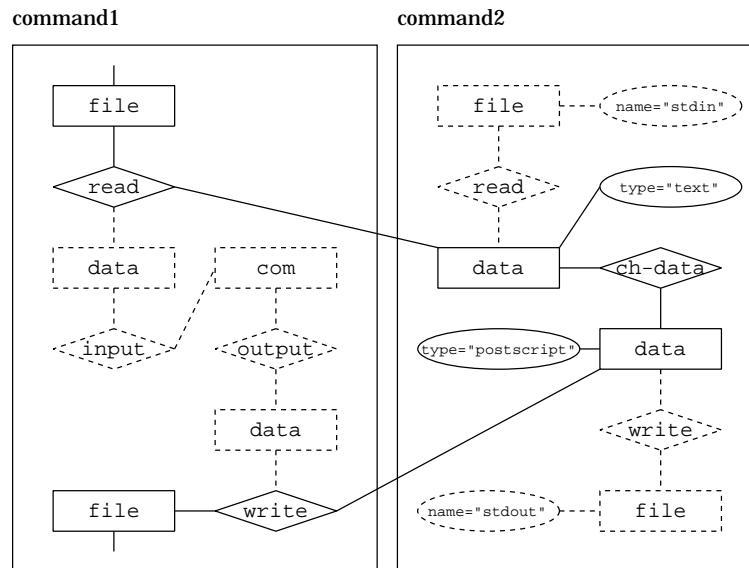


図 4.14: com 型の変換の例 (変換後)

いる。しかし、検索するコマンドのパスを決定する要素が一つだけならば、ほとんどのコマンドは 指定されたパスを前から順番に検索する。例えば、環境変数 `MANPATH` が `usr/local/man:usr/man:usr/lang/man` であれば、コマンド `man` を実行したとき、オプション `M` でパスが指定していなければ、目的のマニュアルファイルを `usr/local/man`, `usr/man`, `usr/lang/man` の順番で検索する。オプションを指定する要素が一つだけかどうか判断するのは、目的のファイルを示す実体から出ている関連 `get-path` の数を数えることにより判断できる。コマンド `man` などは、よく使うマニュアルファイルのある位置は `MANPATH` に記述しておくことが一般的で、目的のマニュアルファイルが `MANPATH` のみによって決定されることは多い。また、`latex` コマンドなどは、目的の検索するパスの記述は環境変数 `TEXINPUTS` のみに委ねられている。目的のファイルのパスを決定する要素が一つだけであることは多いので、パスを決定する要素が一つであれば、目的のファイル名を与え、指定したパスを前から順番に検索するツールを作成することは現実的であると考えられる。しかし、目的のファイル名を得るプログラムを作成することは現実的でないので、「目的のファイルを検索するツール」を作成しようとすると、完全な自動化は現実的でなく、半自動化が現実的な範囲となる。

また、パスを決定する要素が一つだけでも、指定された順番に検索しないコマンドもあるので、そのようなコマンドに対して誤った情報を提供しないよう、どのように対処するか考える必要がある。

第 5 章

UNIX コマンド の制約の実現

Emeraude PCTE では、SDS でサポートされていない制約の実現はプログラミングにより行なう。

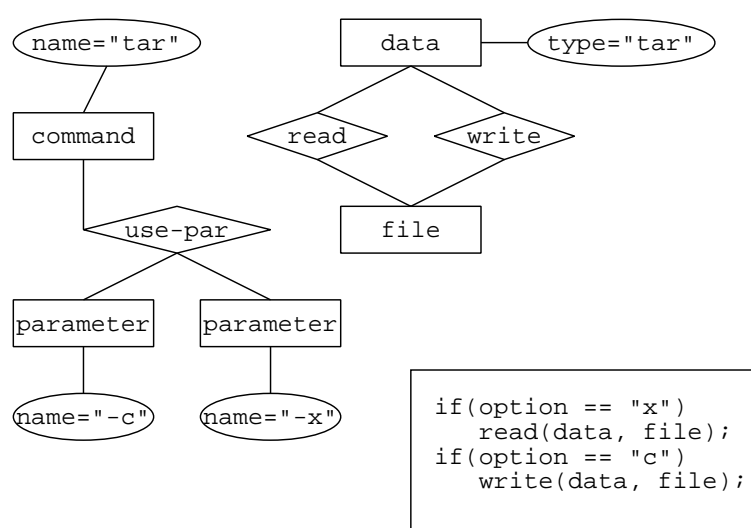


図 5.1: tar のクラス図の一部とアルゴリズム

図 5.1 は `tar` のクラス図の一部である。コマンド `tar` において `type = "tar"` の `data` 型の実体と `file` 型の実体には関係 `read` と `write` が成り立っている。この成り立っている関係はクラス図で表現することができる。しかし、図 5.1 のアルゴリズムのように、`read` と `write` の関係は常に成り立っている訳ではなく、成り立つ為の条件がある。オプション `x` が指定してあれば、`type = "tar"` の `data` 型の実体と `file` 型の実体は `read` しか成り立たず、また オプション `c` が指定してあれば、`write` の関係しか成り立たない。このような

制約はプログラミングを行ない、ツールとして実現する。

4.3.1 節で、`cat`, `man`, `tar`, `lpr` の制約をいくつか紹介したが、制約の記述を追求していく意味でこれらの制約の実現を行なう必要がある。しかし、これらには完全な実現が不可能なものが多数含まれている。

本研究のゴールであるソフトウェアデータベースの確立のためには、そのような制約の実現を実際に試みたうえで、どの程度記述できるかを検討し、ソフトウェアデータベースにおける計算機支援のあり方を考察する必要がある。

以下 UNIX コマンドにおける制約の例を挙げ、具体的な実現の方針を示した上で、実際にツールの作成を試みる。

なお、以下の制約実現の案は、データベース内の情報を参照するものがいくつかあるが、その情報はデータベース上に全て記述済みであると仮定し、情報を参照できない場合のことは考えない。

5.1 機能

`tar` のオプションに関する制約

コマンド `tar` のオプションは `x`, `c`, `r`, `u`, `t` のうちの 2 つ以上を同時に与えることは出来ない。

`tar` の具体的な実行例を与えることにより、そのオプション指定が正しいかどうかを返すツールを作成する。

環境変数 `PAGER` に関する制約

環境変数 `PAGER` では、UNIX コマンド `man` を起動する際、得たマニュアルファイルを環境変数 `PAGER` で指定されている、コマンドによって処理する。

`PAGER` には、ファイルの内容をユーザに読みやすい形に表示するコマンドがセットされるべきだが、`ls` や `wc` などのコマンドがセットされてもシステム上の問題はない。しかし、コマンド `man` がマニュアルを表示する為のコマンドを考えると、それらの値は不適切である。

そこで、環境変数 `PAGER` に適当であるコマンドのリストを表示するツールを作成する。

5.2 実現

tar のオプションに関する制約

tar の排他的なオプションを判断するツールを作成した。このツールは次の手順で判断を行なう。

1. 入力された実行例からオプションを抽出する。
入力された実行例からパーザーにより、オプションを抽出する。
2. オプションが存在するか、データベース上を検索する。
オプションは全て `parameter` 型に属しているが、今回作成した UNIX コマンドデータベースではオプションは全て `option_[オプション名].make_parameter` という名前にした。これを利用し、UNIX コマンド上に存在する tar のコマンドのオプションを検索し、入力された実行例と照らし合わせる。
3. ツール内部のオプション一覧を検索し、さらに同時指定できないオプションが存在していないか判断する。
ツールのプログラミングの際には、そのソースコードにオプションの一覧や、同時指定できないオプションの組合せが記述しておき、これと照らし合わせることで、同時指定できないオプションが実行例の入力の中で同時指定していないか判断する。

以下は tar のオプションに関する制約を実現したツールのソース一部で、入力した文字列が値が、ソフトウェアデータベース上でオプションとして記述しているか否かを返す関数の部分である。

```
int is_supported(ch)
    char ch;
{
    struct objstat current;
    struct linkstat par[LISTSIZE];
    int i, j, k, nb, ret, shortform, opob, same;
    char name[LINKSIZE];
    char *opname;

    ret = FALSE;
    shortform = TRUE;
    opname = "option_";

    getobjstat(".", &current);
    nb = lslinks(".", "make_parameter", NULL, LISTSIZE, par);
    for(i = 0; i < nb; i++){
```

```

linkname(current.o_defid, par + i, shortform, name);
opob = TRUE;
for(j = 0; j < 7; j++){
    if(name[i] != opname[i]){
        opob = FALSE;
    }
}
j = 0;
k = 0;
same = FALSE;

if(opob){
    if(name[7 + j] == ch){
        same = TRUE;
    }
    if(same == TRUE){
        ret = TRUE;
    }
}
}
return ret;
}

```

ここで使用した PCTE ライブラリの関数は、getobjstat と lslinks と linkname である。

- getobjstat(char *path, struct objstat *buf)

オブジェクトの状態を objstat に格納する関数。linkname 関数の引数に オブジェクトの definition Id へのポインタが必要である為、その情報を得る為にここでは用いている。このツールは、tar の情報を格納する dr 型のオブジェクト tar.e に設置しているので、getobjstat(".", ¤t) は、このツールを tar.e で使うことを想定して、tar.e のオブジェクト状態の情報を current に格納する。

- lslinks(char *origin, char *ltype, struct linkstat current, int nb, struct linkstat list[])

指定したオブジェクトからのリンクのリストを得る関数。リンクの総数を返す。ここでは、オプションは parameter 型の実体としてソフトウェアデータベースに記述しているので、lslinks(".", "make_parameter", NULL, LISTSIZE, par) は、tar.e から作成した parameter 型の実体の一覧を得て、linkstat 構造体の配列 par に格納している。

- linkname(struct defid *o_defid, struct linkstat *lstat, int shortform, char l[LINKSIZE])

linkstat 構造体から リンク名を得る関数。lslinks によって得た linkstat 構造体から リンク名を得た。

以下に 実現したツールの実行結果を示す。

```

esh$ op_chk.tool tar xvf foo.tar
option x is supported.
option v is supported.
option f is supported.
esh$
esh$ op_chk.tool tar xvfQ foo.tar
option x is supported.
option v is supported.
option f is supported.
option Q is not supported.
esh$
esh$ op_chk.tool tar xvfc foo.tar
option x is supported.
option v is supported.
option f is supported.
option c is supported.
but c and x is not supported at the same time.
esh$

```

最初の実行例は `op_chk.tool tar xvf foo.tar` と入力したときの例であり、`tar` コマンドを使うとき、`tar xvf foo.tar` と入力したときのオプションをチェックしている。オプションは `x` も `v` も `f` もサポートしているというメッセージを返す。

2 番目の実行例は サポートしていないオプションを入力した例で、オプション `Q` はサポートしていないというメッセージを返す。

3 番目の実行例は 同時指定オプションを同時指定した場合で、`tar xvfc foo.tar` と入力したときのオプションをチェックしている。その結果 オプション `x`, `v`, `f`, `c` はそれぞれサポートしているが、`x` と `c` は同時指定できないというメッセージを返している。

環境変数 PAGER に関する制約

`command` 型に与えた属性 `type` を参照し、環境変数 `PAGER` に適するかを判断するツールを作成した。次の方法で `PAGER` に適するコマンドかを判断するのに利用することができる。

- 環境変数 `PAGER` に適するコマンドのリストを参照する。
- 属性 `type` の値が “`pager`” である `command` 型の実体を列挙する。
- UNIX コマンドデータベース内の関係を参照し、標準入力により得たデータを標準出力に出力する関係の構造 (図 5. 2) を持っているものを列挙する。

以上のそれぞれの作業によって得られる `command` 型の実体の集合の和集合をとり、最終的に `PAGER` に適するコマンドの集合として表示する。

3 つの作業の和集合を取るのは、いずれも得られる情報が不十分だからである。例えば、図 5. 2 の `pager-relation` も `PAGER` に適する全てのコマンドがこの構造を持つ訳ではない。ま

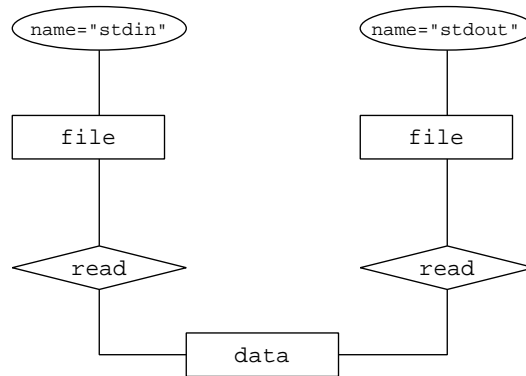


図 5.2: PAGER に適するコマンドが持つと考える関係

た、3つの作業により得られるそれぞれの `command` 型の実体の集合の和集合も全てを網羅している訳ではなく、情報量は十分でない。

しかし、環境変数に PAGER にセットするべきコマンドを知らないユーザが参照する情報としては有効である。

次のソースは `command` 型の実体の属性 `type` を参照し、PAGER にセットするコマンドとして適切であるか判断するツールのプログラムである。

```

attrval val;
char buf[100];
char *com_obj;

val.v_string = buf;
getattr(com_obj, "type", 100, val);
if(strcmp(val.v_string, "pager") == 0){
    printf("%s is guessed right value for PAGER.\n", argv[1]);
}

```

このツールで新たに使用した PCTE ライブラリの関数は `getattr` である。

- `getattr (char *path, char *attr, int size, attrval value)`

オブジェクトの属性を返す関数。 `attr` は共有体であり、属性 `type` は `string` 型なので、 `val.v_string` で `type` の値を得ることができる。ここで代入した `com_obj` は参照する `command` 型のオブジェクトの相対ス名が格納されている。

次にこのツールの実行結果を示す。

```

esh$ pager_check1.tool less
less is guessed as right value for PAGER.
esh$
esh$ pager_check1.tool cat

```

```
cat is guessed as right value for PAGER.  
esh$  
esh$ pager_check1.tool man  
man is guessed as wrong value for PAGER.  
PAGER expects the type "pager".  
esh$  
esh$ pager_check1.tool hello  
command hello is not supported.  
esh$
```

この実行結果は、まず `cat` が環境変数 `PAGER` として相応しいかチェックしている。`cat` の `type` は `pager` なので、この場合は `PAGER` として適切であるという結果を表示する。次に `man` が環境変数 `PAGER` として相応しいかチェックしている。`man` の `type` は `pager` ではないので、この場合は `PAGER` として相応しくないという結果を表示する。最後に `hello` が環境変数 `PAGER` として相応しいかチェックする。`hello` は UNIX コマンドデータベース上にコマンドとして記述していないので、サポートしていないという結果を表示する。

5.3 評価

今回、作成したデータベースは関連を記述した UNIX コマンド、実現した制約、いずれも量が不十分だったので、構築実験の結果が思うように得られなかった。

本研究で制約をツールとして実現していった過程の範囲では、制約のプログラミングを行なうには、実体間の関係を形式的に記述することが有効であるということが推測できる。なぜなら、制約の対象となるオブジェクトを参照するには、関連をたどることにより、オブジェクト名を得ることが多かったからである。

その反面、形式的な記述には限界があり完全な実現が不可能であることも推測できる。そのような制約の本来の意味を記述するには、非形式的な記述に頼らざるを得ないが、本研究で作成した UNIX コマンドデータベースは形式的な記述のみを行なっているので、制約本来の意味を表現できていない。例えば、本研究で作成したデータベースから環境変数 `PAGER` に関して「テキストファイルをユーザに読みやすい形で表示する。」という本来の意味は記述できておらず、`command` 型に属性 `type` を与えるなどして、近似を行なっている。

また、本研究は作成するスキーマを一つだけとし、それをメタ情報として定義した。メタ情報を定義することにより、複数の UNIX コマンドの関係図を形式的に記述することができる。本研究で作成した、環境変数 `PAGER` の制約を実現するツールの一部において `command` 型の `type` を参照するが、この際、メタ情報を定義しておいた為、どの `command`

型をチェックするときも、同じ作業でチェックすることができた。このことにより、メタ情報の有効性が UNIX コマンドデータベースにとって必要なものであることが推測できる。

本研究では階層構造を用いて、メタ・クラス図、クラス図、インスタンス図を作成することを想定しているが、作成した UNIX コマンドデータベースは設計の上でインスタンス図を作成することを考慮に入れていない為、インスタンス図を記述するとしたら、どのように記述するかという問題が未解決のままとなっている。

第 6 章

考察

6.1 評価

- メタ・クラス図で定義した実体、関係に沿って形式的に記述している為、“ファイルを読む”、“環境変数を参照する”など、動作の目的に応じた検索を行ないやすい。
- ソフトウェアオブジェクトの関係を ERA 図で記述することにより、コマンドに関わるオブジェクト及び、その役割の全体像を視覚的に理解することができるが、現段階において PCIE では生成したオブジェクトを視覚的に表示する機能はない。しかし、PCIE による UNIX コマンドデータベースでは ERA 図の持つ情報を形式的に記述しているので、今後、視覚化するソフトウェアを開発することにより、ERA 図を得ることのできる可能性を持っている。
- 全ての制約をツールとして完全に実現することはできない為、そのような制約の本来の意味を伝えようとする、非形式的な記述が必要。例えば、“環境変数 PAGER にはテキストファイルをユーザに見やすい形で表示する”という意図は形式的な記述では完全に表現できず、近似を行なうしかない。
- 制約はいくつかの種類に分類することができる。まず、プログラムで実現可能な制約と部分的な実現が可能な制約に分けることができ、部分的な実現が可能な制約はさらにいくつかに分かれる。現段階においては次の 3 つが確認できている。
 - － 位置情報に関する制約
 - － 意図に関する制約
 - － 動的情報に関する制約

これらは各々において同じような方法で、部分的解決ができる。

6.2 制約の分類

6.2.1 分類結果

UNIX コマンドには計算機上で、制約の記述が難しいものが存在する。最初、制約は難しさによって分類できる。

- 実現可能
- 部分的な実現が可能

さらに、「部分的な実現が可能」な制約は、その実現方法によって分類できる。いくつかのコマンドを調べた結果、図 6.1 のように分類を行なった。

種類	例	(部分的な) 解決方法
位置情報に関する制約	環境変数 MANPATH の設定 環境変数 TEXINPUTS の設定 環境変数 PATH の設定 gc c コマンドのオプション -L や -I	管理者が位置情報を記述 find により検索 他のユーザの .cshrc を検索
プログラムの意図に関する制約	環境変数 PAGER の設定 環境変数 TROFF の設定 環境変数 TCAT の設定	適切なもののリストを作成する コマンドの関係の記述を参照 意図に関する型を導入
動的情報に関する制約	環境変数 TCAT の設定 tar のオプション z や Z	インスタンス図の作成

図 6. 1: 制約の分類例

- 位置情報に関する制約 … MANPATH 等のパス名に関する制約。サイトの変化等に y り、適切なパス名が変かするという難しさがある。しかし、インストール後に一意に決まり、かつ形式的に表現できる。
- プログラムの意図に関する制約 … PAGER 等の、要求するコマンドに関する制約。要求の意図はプログラミングが難しいという難しさがある。

- 動的情報に関する制約 … ファイルデータや環境変数などに対する、実行されるまで分からないものに対する制約。具体的なオブジェクトに関する制約なので、インスタンス図を参照する必要があるという、難しさがある。インスタンス図に対する問題点は 4.4.1 節で説明した。

ここで注意をしておくと、これらの分類により得られるそれぞれの制約の集合は非分離である。例えば環境変数 `TCA T` は、`man` コマンドでオプション `-t` をつけたとき、マニュアルファイルを環境変数 `TR OFF` で処理したデータをさらに処理するコマンドを指定した環境変数だが、この `TCA T` にセットされるコマンドは動的に決まる。例えば、環境変数 `TR OFF` に `nroff` がセットされていれば、`TCA T` が処理するデータは、テキスト形式なので、「テキストファイルを見やすい形で表示する」コマンドが `TCA T` にセットされるべきだが、`TR OFF` に `psroff` がセットされていれば、`TCA T` が処理するデータは、`postscript` 形式なので、「`postscript` ファイルを見やすい形で表示する」コマンドが `TCA T` にセットされるべきである。これら両者の制約は意図に関する制約である。

環境変数 `TCA T` に関する制約は、意図に関する制約と動的情報に関する制約、両方の性質を持つ。

6.2.2 C のプログラムとの比較

前節で UNIX コマンドでの制約で部分実現が可能なものを位置情報に関する制約、意図に関する制約、動的情報に関する制約に分類した。仮にこれを UNIX コマンドではなくて、C 言語によるプログラムのデータベースを作成した場合を考えてみる。

位置情報に関する制約

位置情報に関する制約として、例えば C 言語をコンパイルする際には、`include` するファイルやライブラリの位置を示す、オプション `-I` や `-L` が一般的に必要である。

UNIX コマンドにおける位置情報に関する制約は、他の人の `.cshrc` を検索、`find` により検索、位置情報を管理者が記述といった方法で部分的に実現が可能であったが、C 言語プログラムのデータベースにおいても、`find` により検索、位置情報を管理者が記述といった方法は、UNIX コマンド同様、部分的な実現が可能である。また、UNIX コマンドで、`.cshrc` を検索するかわりに、`Makefile` を検索する方法があるが、それが個人の目的や環境に応じたものであるかを判断することが困難であることは `.cshrc` と同じである。

意図に関する制約

例えば、関数 `qsort` は

```
void qsort (base, nel, width, compar)
char *base
int nel, width
int (*compar)()
```

という形をしているが、`compar` には大小を比較する関数へのポインタが入る。しかし、「大小を比較する」という意図はプログラミングが難しいため、`qsort` の引数に関する制約をプログラミングで完全に実現するのは不可能である。

前節では UNIX コマンドにおける意図に関する制約は、型の導入、関係の参照、適切なもののリストを作成 といった方法で部分的な実現が可能であったが、C 言語 の `qsort` の場合でも同様の方法で部分的な実現が可能である。

動的情報に関する制約

C 言語のプログラムはコンパイル時には何の問題もないが、実行時に初めて出るエラーがある。例えば、0 で割算をしたり、NULL ポインタにアクセスしたりといったようなことが、それに当てはまる。その解決方法は プログラム上の関数や変数の関係を理解し、具体的な値を代入してみることで解決できる。これが、UNIX コマンドデータベースでのインスタンス図の作成にあたる。

しかし、これを実現しようとするすると作成した C 言語のプログラムの動きを実現するプログラムの作成が必要であり、プログラムの動きとはプログラムそのものであるので、プログラムのインスタンス図を作成するプログラムの中に元のプログラムを組み込むという事態に陥り、現実的でないという点で UNIX コマンドの場合と一致している。

以上より、本論文により得た分類は C 言語プログラムのデータベースにも適用できることが言えた。結論として言えることは、本研究により得た分類は UNIX コマンドに限定したものではない。

6.3 メタ情報

UNIX コマンドデータベースの作成の際にはメタ情報が必要であったが、C 言語プログラムのデータベースでも必要であるかをここで考える。

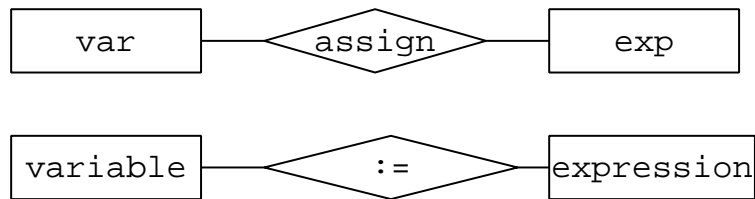


図 6.2: メタ情報を用いない不統一な記述例

例えば、変数に式を代入している箇所を検索したいとき、変数を表わす実体と式を表わす実体の間に代入という関係が成り立っているものを検索することで可能である。しかし、図 6.2のように、変数と式の間に代入するという記述の仕方が2通り以上あると、検索に不便である。例えば上図の例では「var と exp の間に関係 assign」と「variable と expressionの間に関係 :=」の2通りを考える必要がある。しかし「var と exp の間に関係 assign」をメタ情報とすると、検索にはこの場合だけを考えればよい。

C 言語プログラムのデータベースでも、メタ情報は必要である。

第 7 章

おわりに

7.1 まとめ

本研究では、ソフトウェアデータベースの構築方法の確立のために UNIX コマンドデータベースの構築実験を行なった。データベースの構築方針としては、ソフトウェアオブジェクト間の制約・関係を形式的に記述する。

まず、関係を記述する手段として、メタ・クラス図、クラス図、インスタンス図を持つ階層構造を採用した。複数の UNIX コマンドを統一的に記述する為にメタ・クラス図を用い、予期しない実行結果に対処する手段として具体的なオブジェクトを参照する手段としてインスタンス図を用い、クラス図は具体的なオブジェクトを決定する前の静的な関係を参照する手段として用いる。

データベースとして実現するコマンドとして、`cat`, `man`, `tar`, `pr` を選択した。各々のコマンドに対して関係・制約の抽出を行ない、関係をクラス図として記述した。クラス図を作成するにはメタ・クラス図が必要であるが、メタ・クラス図は各々の UNIX コマンドの関係の抽出の際に順次、変更・拡張を行なった。この際、統一性を失わないように気をつけた。また、抽出した制約はプログラミングを行ないツールとして実現した。

以上の過程を経て作成した UNIX コマンドから制約の分類を行ない、分類ごとの実現方法を示した。制約には実現可能なものと、部分的な実現が可能なものが存在し、部分的な実現が可能なものはさらに、位置情報に関する制約、意図に関する制約、動的情報に関する制約に分類した。また、この分類を用いて C 言語のプログラムのデータベースを作成することを想定し、得られた分類が UNIX コマンドに限定したものではないことを確かめた。

7.2 結論

制約・関係の記述は難しい

本研究では、UNIX コマンドにおける制約・関係を形式的に記述することを試みたが、その上で様々な問題が生じた。非形式的なものの記述は難しいとされているが、形式的な記述を行なうことでさえ難しいことが確認できた。

形式的な記述は必要

本研究により、制約・関係を形式的に記述することは、目的に応じた検索を行なう為に必要であることを確認した。

非形式的な記述も必要

制約には形式的に記述できないものも存在し制約本来の意味から近似を行わなくてはならない為、制約本来の意味を伝えるには非形式的な記述も必要であることを確認した。開発者の意図をユーザに伝えることはユーザがコマンドを理解する上で必要なので、ソフトウェアデータベースでは非形式的な記述も扱う必要がある。しかし、それをどのように扱うかという問題に対しては未解決である。

メタ情報が必要

また、UNIXコマンドデータベースにおいて、効率良く検索を行なうには複数のUNIXコマンドの間で共通の定義となるメタ情報が必要であることを確認した。また、これをC言語のプログラムのデータベースと比較することにより、メタ情報が必要であるのはUNIXコマンドに限定したことではないことを確認した。

制約の分類が可能

また、制約はいくつかの種類に分類できることが確認できた。本研究では、まず制約を形式的な記述で実現可能な制約と部分的な実現が可能な制約に分類した。部分的な実現はさらに、位置情報に関する制約、意図に関する制約、動的情報に関する制約に分類したが、これをC言語のプログラムのデータベースと比較することにより、UNIXコマンドに限定したことではないことを確認した。

本研究を進めていくことは有効

この分類により、ソフトウェアデータベースの制約の細分化を行なうことができ、ソフトウェアデータベースの制約という大きな問題を小さな問題に分割して考えることができる。小さな問題を一つ一つ取り組んでいくことで、ソフトウェアデータベースの構築方法の糸口を得ることが期待できる。

ソフトウェアデータベースの制約の細分化という点で本研究を進めていくことは有効である。

7.3 今後の課題

- メタ・クラス図や制約の分類の確立
さらに いろいろなコマンドの制約・関係を実現することにより、UNIX コマンドのメタ・クラス図や制約の分類を確立する。
- 細分化した問題の考察
制約の分類により細分化した UNIXコマンドデータベースの問題を考察すると共に、問題の細分化の有効性、ソフトウェアデータベースの問題にどの程度適用できるかを検証する。
- インスタンス図の半自動作成ツールの設計、実装
動的情報参照の部分的実現を試みる。

謝辞

最後に、本研究を行なうに当たり終始 御指導頂きました権藤克彦助教授には深く感謝致します。

また、本研究に関して多くの貴重な御意見を頂いた片山卓也教授 はじめ ソフトウェア基礎講座の皆様に深く感謝すると共に厚く御礼を申し上げます。

参考文献

- [1] 矢上 克之：UNIX ファイルの依存解析，東京工業大学卒業論文，1997
- [2] 今井 久夫：ER モデルによる UNIX コマンドデータベースの作成，東京工業大学卒業論文，1998
- [3] PCTE : A Basis for a Portable Command Translator for UNIX Shell Functionality, 1988
- [4] ER MODEL FOR THE UNIX COMMAND ENVIRONMENT (PC) COMMAND ANALYSIS AND BINDING, 1995
- [5] 鯨坂 恒夫, 沢田 篤史, 満田 成紀：Eraude PC, コンピュータソフトウェア vd.10 ソフトウェア評論，1993
- [6] 山口 和紀 監修：The UNIX System Text, 技術評論者
- [7] 沢田 篤史, 鯨坂 恒夫, 松本吉弘：実体・関連モデルへの意味的制約の統合と PC データモデルへの展開