修 士 論 文

SPIN を用いたバイナリモデル検査

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

土肥 雅俊

2008 年 3 月

修 士 論 文

SPIN を用いたバイナリモデル検査

指導教官　青木利晃 特任准教授

審査委員主査　青木利晃 特任准教授
審査委員　片山卓也 教授
審査委員　鈴木正人 准教授

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

0610060 土肥 雅俊

提出年月: 2008 年 2 月

概 要

本研究では、形式的検証手法の 1 つであるモデル検査を利用した新たな検査手法であるバイナリモデル検査について提案する。

モデル検査を利用して性質の正当性を証明する場合、検査対象をモデル化し、その正当性を証明する。つまり、プログラムを対象とする検査の場合は、プログラムの振る舞いを適切に抽象化し、モデルとして表現することが重要となる。しかし、プログラムの振る舞いをモデル化することは困難である。そこで本研究では、Spin の C 言語埋込み機能を利用した C プログラムのバイナリモデル検査手法を提案する。バイナリモデル検査とは、本研究独自の検査手法である。検査対象 C プログラムのバイナリが使用しているメモリの一部を状態とみなしてモデル検査を行うため、このように名付けた。実際にバイナリが動作しているメモリを監視する事によってプログラムの運用環境に近い条件での検査が可能となる他、状態遷移モデルの一部をバイナリ実行結果から自動で構築し検査にかかる作業を減少させるなどの利点がある。また、本研究ではバイナリモデル検査実装のためにモデル検査ツール Spin の C 言語埋込み機能を利用している。この機能は、Spin 用の言語である Promela に C 言語の要素を埋込み、検査用のモデルを作成することを可能にする機能である。

本論文前半では、本研究の前提となる知識を簡単に説明し、シンプルな構造であるソートプログラムに対してバイナリモデル検査を適用して手法の概要を示した。さらに、発展研究としてソートログラムよりも規模の大きい自作 ls プログラムに対し、バイナリモデル検査を適用しその有効性について考察した。これらの実験により、本研究の適用法、有効性、問題点などが明らかとなった。

本論文後半では、実験結果を元に手法の整理を行い、これまでの研究に対するまとめと考察を行った。現段階では、いくつかの制約のもとバイナリモデル検査を実行する必要がある。この制約について考察を行い、明らかとなった問題については、今後の課題として予想される解決法と共に示した。

目 次

図 目 次

第1章　はじめに

近年、ソフトェアの信頼性確保のために形式的検証手法が注目されている。形式的検証手法の1つであるモデル検査では、検査対象をモデル化し、その正当性を証明する。つまり、プログラムを対象とする検査の場合は、プログラムの振る舞いをいかに適切に抽象化し、モデルとして表現するかが重要である。しかし、プログラムの振舞いをモデル化することは困難である。例えば、モデル検査ツール Spin[J.H05] を用いてプログラムを検査する場合には Promela 言語を用いて対象の振る舞いを記述する。ここで、問題点が2つある。1つ目は、モデルが単純すぎれば検査対象が失われてしまい、詳細にすれば状態爆発が頻繁に発生すること、2つ目は、プログラムを Promela(PROcess MEtamodel LAnguage) に直接変換し、モデルを記述しようとすると言語相違からモデルの振舞いが変わってしまうことである。例えば、C 言語にあるポインタや構造体などは Promela にはない。これらの問題は、モデル検査の行程を困難にしている。たとえ、経験豊富な技術者であってもプログラムの振舞いや性質を適切に捉え、これらの問題を解決することは容易ではない。そこで、本研究では、Spin の C 言語埋込み機能を利用した C プログラムのバイナリ検査手法を提案する。Spin は Promela 言語を採用しているが、C 言語を直接、検査コード内に埋込む機能も有している。その機能を利用し、検査対象プログラムに大きな変更を加えずに、適切な検査モデルを作り出す手法を提案する。そして、本手法が適用できる条件を整理し、検査手法を系統化、提案手法の有効性や応用法を評価・考察する。

第2章　SPIN とバイナリモデル検査

2.1　モデル検査とは

モデル検査（Model Cheking）とは、形式的検証手法のひとつである。形式的検証手法では、数学的・論理的基盤に基づいてある性質の正しさを証明する。モデル検査では、検査対象の状態遷移モデルを有限オートマトンに対応付け、ノードとエッジからなる有向グラフで表現する。そして、有限範囲でグラフの遷移を網羅的に全自動探索することで調べたい性質の正しさを保証する。しかし、実世界の問題を扱おうとすると、モデル検査は状態爆発の問題に直面する。実世界を詳細にモデルに表現しようとすると、遷移する状態数が膨大な量となり検査不能となってしまうからである。この問題を回避するため、モデルを作成する際には注目するべき問題を見極め、適切に検査するモデルを抽象化する必要がある。

2.2　モデル検査ツール SPIN

モデル検査ツール SPIN とは、ソフトウェアをモデル検査するためのツールである。SPIN は、仕様記述言語 PROMELA による記述を入力として表明、到達性、進行性、LTL で書かれた性質などの検査を自動的に行うツールである。PROMELA は、並行動作する有限オートマトンを記述することに長けている言語である。C ライクな言語であり、代入、演算、if 文、do 文、goto 文などの命令を持ちいて並行プロセスを記述する。一方で、PROMELA は、ポインタや浮動小数点型、二次元配列などのデータ型がない。しかし、代表的なプログラミング言語ではこれらのデータ型を使用していることが多く、その振舞いを PROMELA のみで適切に表現、検査する事は困難である。そこで本研究では SPIN の C 言語埋込み機能を用いて、すでにあるプログラムコードをそのまま埋込み、さらに、プログラムの振舞いをより簡単、適切にモデル化する手法を提案する。

2.2.1　埋込 C コードについて

SPIN は、バージョン 4.0 以降で C コードを PROMELA に埋め込む機能をサポートしている。この節では、この機能について説明していく。埋め込み C では、以下の 4 つの式を提供している。

c_expr, c_code, c_decl, c_state, c_track

これらの式は、任意のCコードに適用することが可能であり非常に強力だが、その反面、自由度が高くモデル検査実行に予期せぬ影響を与える事がある。また、式に埋込んだCコードのシンタックスチェックを構文解析から検査実行までSPINは一切行わない。そのため、検査途中にエラーが発生し検査結果が得られない場合もあり、ユーザはその使用に注意を払う必要がある。

埋込んだCコードは、検査器によってPROMELAモデルの一部と見なされ遷移する状態として扱われる。式c_codeの実行中は、他の遷移に割り込まれることはなく、PROMELAが提供するd_stepのようにアトミックに実行される。よって、通常SPINはc_code内のC変数の変化を知る事はできない。式c_exprは、ユーザが定義できる、booleanに関するガードである。式c_declとc_stateは、様々なC言語のデータ型やデータオブジェクトの宣言を取り扱う。宣言したものを状態ベクトルの一部とすることができる。c_trackは指定したデータオブジェクトの値をトラックする。トラックするデータオブジェクトは、どこで宣言されたものでもよい。各式について詳細に説明する。

c_code　ー　埋込Cコードフラグメント

- 文法
  - c_code { /* c code */ }
  - c_code '[' /* c exprt */ ']' { c code */ ;}
- 機能
  - 定義されたCコードを丸括弧の中に記述できる。
- 説明
  - c_codeは、Cコードフラグメントを PROMELAモデル内で使用することをサポートする。使用法は２種類ある。１つ目は、四角括弧を使用する書き方で、四角括弧内でCの評価式が判定される。結果が0以上ならば、c codeが実行される。結果が0ならば、丸括弧内のc codeは実行されず、反例が出力される。
- 例
  - １つ目の例では、初めにPROMELAのint型変数qを宣言している。qは、自動的に検査器の内部ステイトベクター(nowと呼ばれる)に組み込まれる。Cの大域変数としてint型へのポインタ変数pも宣言している。pは、ステイトベクター外にありSPINはpの変化を監視しない。つまり、pは通常の大域変数として振る舞うが、検査器はその状態をトラックしないということである。もし、

p の値の変化を監視したいならば、以後に説明する c_decl,c_state,c_track 式を利用する。
２つ目の c_code では、特別な接頭辞を使う事なく、p を直接参照している。これは、p がステイトベクターの外にあるからである。３つ目の c_code では、メモリエラーが発生しないよう四角括弧内で p の値が 0 以外であることをチェックしている。判定が真であれば、c code が実行され、p に PROMELA 変数 q のアドレスが代入される。このとき、q はステイトベクター内にあるので特別な接頭辞 now をつけなければならない。最後の c_code では、あらかじめ予約された変数_pid(プロセス init のプロセス id) の値を表示する。また、ここで見られるようにプロセスの局所変数にアクセスする場合、P+プロセス名 → 変数名というように記述する。

```
1 int q;
2
3 c_code{ int *p };
4
5 init{
6 c_code{ *p = 0; *p++; };
7 c_code [p != 0] { p = &(now.q); };
8 c_code {Printf("%d\n",Pinit ->_pid); }
9 }
```

図 2.1: c_code 使用例

– ２つ目の例は、c_code の事前条件チェック (c_code '[' /* c exprt */ ']' { c code */ ;} の c exprt の部分) を利用して境界条件のチェックをする例である。この例は、プロセス ex のローカル変数 ptr が指す配列 x に値を代入する操作のチェックをしている。

```
1 c_code [ Pex ->ptr != 0 && now.i < 10 && now.i >= 0] {
2          Pex ->ptr.x[now.i] = 12;
3 }
```

図 2.2: 境界条件と NULL ポインタのチェック

この例の事前条件チェックではまず、プロセス ex のローカル変数 ptr が NULL ポインタでないかを調べている。さらに、配列の要素番号 now.i が配列のサイズ

0〜10 の境界を満たしているのかを調べている。この事前条件の真偽が真ならば、配列 x に 12 が代入されるはずである。このように c_code の事前条件チェックを利用すればポインタの NULL ポインタチェックや配列の境界条件チェックが可能となる。バイナリモデル検査においてもプログラムの静的検査の簡易版的位置づけて利用する。

c_decl,c_state,c_track　ー　埋込み C のデータ宣言

- 文法
  - c_decl { /* c declaration */ }
  - c_state string string [string]
  - c_track string string
- 説明
  - c_decl,c_state,c_track は、プロセス宣言の外である大域領域においてのみ使用できる。

    c_decl 内でデータ型を定義すると、作成されるコード内のどこであっても使用できるようになる。c_code 内でもデータ型の宣言は可能であるが、宣言した変数をステイトベクターに追加したいならば c_decl を使用してデータ型の宣言を行う必要がある。

    c_state は、大域領域で使用され、2 つあるいは 3 つの引数を持つ。1 つ目の引数は、データ型とその変数の名前である。2 つ目は、変数のスコープである。3 つ目には、変数の初期値を設定する。スコープは、3 種類あり、Global,Local,Hidden である。Global を指定すると、その変数は大域変数として扱われる。Local を指定すると、その変数は、さらに指定したプロセスの局所変数として扱われる。Hidden を指定すると、その変数は大域変数として扱われるのだが、Spin の内部ステイトベクターの外に宣言される。

    C_TRACK は、大域領域で使用され、状態として、あらゆる変数やメモリ領域を指定することができる。第一引数は、監視したい領域の先頭アドレスである。第二引数は、監視したい領域のサイズである。

    これらのデータ宣言フラグメントを使用する際に注意するべきことがある。埋込み C 機能内で宣言する型の名前が Spin 内部で検査用に使用されている名前と衝突しないように注意しなければならない。Spin は、この問題に感知しないので自己での判断が必要となる。実際、stack や P0,P1,Q0,Q1 などの名前の C 変数を埋込 C 上で使用すると Spin の内部変数と衝突が起こる。
- 例

– この例では、c_decl,c_code,c_state での宣言方法を示している。宣言の仕方によって、変数がステイトベクター内に置かれるかどうかが変わってくる。

```
 1 c_decl {
 2   typedef struct Proc {
 3     int state;
 4   } Proc;
 5
 6   typedef struct Rendez {
 7     int lck;
 8     Proc *p;
 9   } Rendez;
10 }
11
12 c_code{
13  Proc H1;
14  Rendez RR;
15 }
16
17 c_state "Rendez R1" "Global"
18 c_state "Rendez R2" "Local" "now.R1"
19
20 c_state "extern Proc H1" "Hidden"
21
22 c_decl {
23  #include "types.h" /* declare type Rendez */
24 }
25
26 c_track "&RR" "sizefo(Rendez)"
27
28 active Proctype ex2(){
29  c_code { now.R1.lck = 1; };   /* global */
30  c_code { Pex2->R2.lck = 0; }; /* global */
31  c_code { H1.state = 2; };     /* C */
32 }
```

図 2.3: c_decl,c_state,c_track 使用例

c_expr　ー　埋込み C の条件式評価

- 文法
  - c_expr { /* c code */ }
  - c_expr '[' /* c expr */ ']' { c code }
- 機能
  - 丸括弧内には、C 言語の意味論に基づいた C コードが記述される。その C コードの評価に際して、副作用 (変数の変化) があってはならない。
- 説明
  - c_expr は、PROMELA 内でガード条件として使用される。

    c_expr の使用法は、２種類ある。１つ目は、四角括弧がない場合である。この場合、丸括弧内の一般的な C の評価が実行され、その結果が 0 以外の場合に真となる。２つ目は、四角括弧を丸括弧の前で使用する場合である。まず、四角括弧内の評価が実行され 0 以外で真となれば、丸括弧内のコードがさらに評価される。四角括弧内の評価結果が 0 となり偽となった場合は、assertion が実行される。また、２種類いずれの場合も丸括弧内の評価結果が 0 となり偽となれば、そこで実行が停止し、ブロックされる。また、括弧内の評価式において変数の変化はあってはならない。なぜならモデル検査実行の際に評価式は何度も呼ばれる可能性があるからである。SPIN は評価式中の変数の変化は監視できないため、バックトラックで再度この評価式に至ったときに必ず同じ変数値であるためには、評価式中の変数の変化があってはならないのである。
- 例
  - この例では、do ループ内に４つの非決定的な遷移がある。初めの２つは評価式であり、２つの違いは、同じ局所変数 x を埋込 C 機能を使用して評価するか、PROMELA を使用して評価するかだけである。

    ３つ目のガードでは、fct() という C 関数が返す int 型の値を評価している。この関数 fct() は、c_code を使用して大域領域のどこにでも定義できる。

```
 1 active proctype ex3()
 2 {
 3 int x;
 4
 5 do
 6 :: c_expr{ Pex3 ->x < 10 } ->
 7         c_code { Pex3 -> x++; }
 8 :: x < 10 -> x++
 9 :: c_expr { fct() } -> x--
10 :: else -> break
11 od
12 }
```

図 2.4: c_expr 使用例

2.3　バイナリモデル検査とは

2.3.1　概要

バイナリモデル検査とは本研究で新たに定義した検査手法のことである。検査対象プログラムをコンパイルしてバイナリ実行し、同時に検査プログラムを開始する。検査プログラムは、バイナリが使用しているメモリの一部を状態空間とみなしてモデル検査を行い(図 2.5)、探索していく。このような手順を本手法では取る。そのような理由からバイナリモデル検査と名付けた。また、従来のデバッガとは異なりブレイクポイントごとに監視を行うのではなく、実際にバイナリが動いているメモリ領域に対して網羅的に自動探索を行える点も大きな特色である。これにより、プログラム運用環境により近い状態で検査が可能となる。(図 2.6)

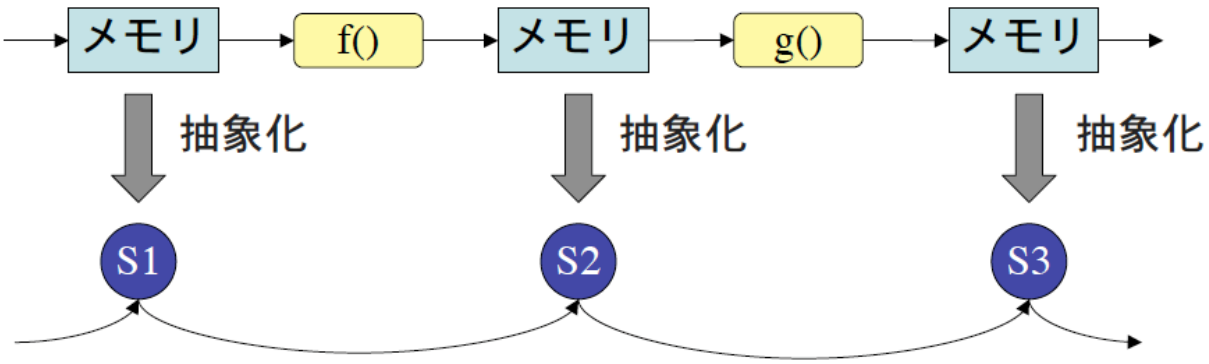

図 2.5: バイナリモデル検査のイメージ

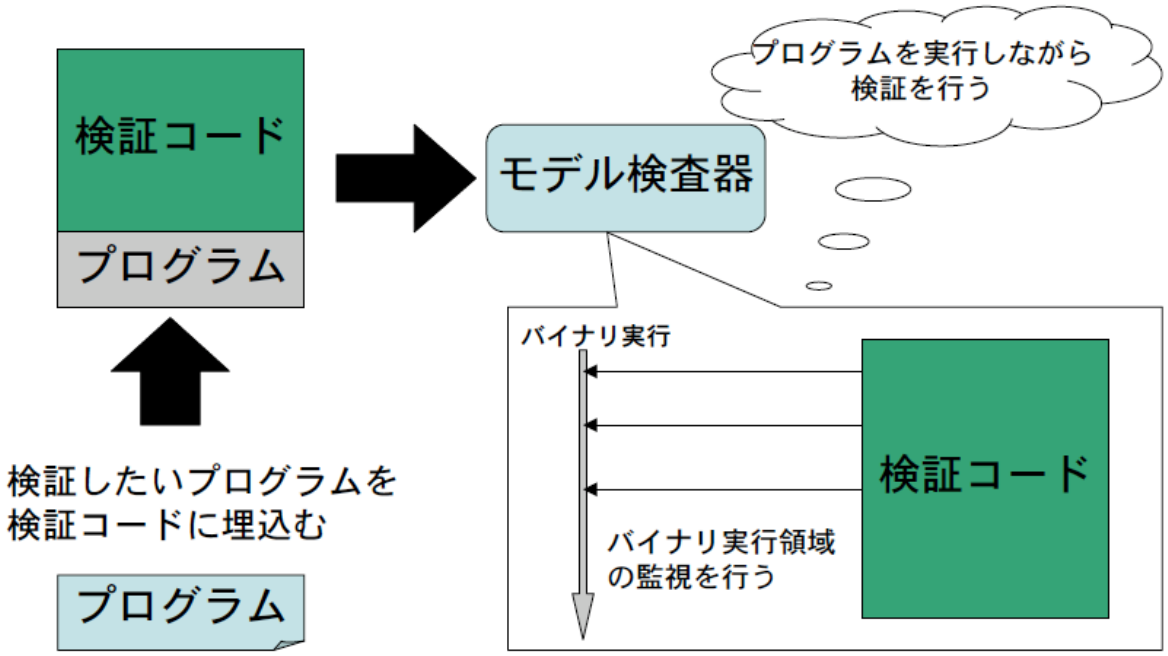

図 2.6: バイナリモデル検査の流れ

2.3.2　実際の手順

本研究では、バイナリモデル検査に SPIN を使用するので、図 2.7 のように手順は一般的な SPIN を用いたモデル検査の手順と同じである。ただ、反例を出力する際のコマンドが若干異なるので示しておく。図 2.8 において、初めの 3 つのコマンドは通常のモデル検査と同じである。異なる箇所は、反例を出力するコマンド pan のオプション-C である。通常 SPIN で反例を出力する場合には、spin コマンドで オプション t を呼び出し、trail ファイルを元に反例を SPIN が出力する。しかし、SPIN は PROMELA で記述した部分しか反例実行しないので埋め込んだ C プログラムがどのように実行されて反例に至ったかがよくわからない。そこで、pan コマンドに提供されているオプション C を用いる。このオプションを指定して pan を実行すると、pan は trail ファイルをもとに埋め込んだ C プログラムを実行しながら反例を出力する。バイナリモデル検査では、C の実行結果を知りたい場合が多いので反例を解析する場合こちらの pan -C コマンドを利用する方が多い。図 2.8 は、実際にコマンドを打ち込んだ例である。

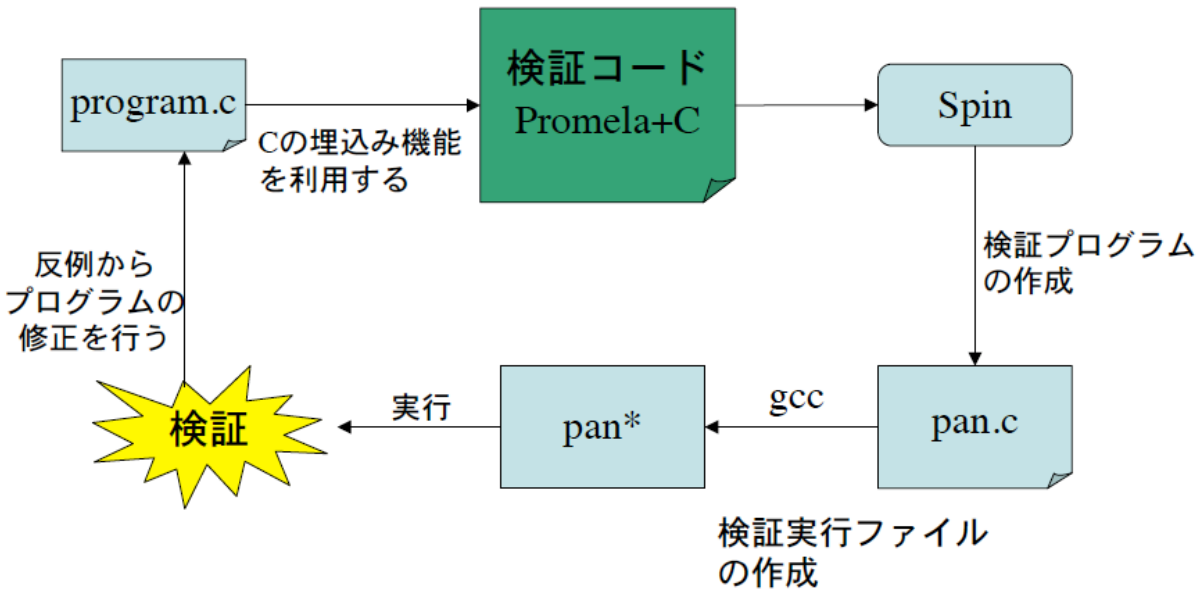

図 2.7: バイナリモデル検査　実際の手順

```
[masatoshi@jaist] %spin -a example.spin
[masatoshi@jaist] %gcc -o pan pan.c
[masatoshi@jaist] %./pan
hint: this search is more efficient if pan.c is compiled -DSAFETY
pan: 0 <= Ptest->i && Ptest->i < 5 (at depth 1553)
pan: wrote example.spin.trail

(Spin Version 5.1.1 -- 11 November 2007)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim          - (none specified)
assertion violations +
acceptance   cycles  - (not selected)
invalid end states +

State-vector 12 byte, depth reached 1552, errors: 1
     1553 states, stored
        6 states, matched
     1559 transitions (= stored+matched)
        0 atomic steps
hash conflicts: 0 (resolved)

2.501     memory usage (Mbyte)


pan: elapsed time 0.01 seconds
[masatoshi@jaist] %./pan -C
                  .
                  .
               反例出力
```

図 2.8: バイナリモデル検査　実際のコマンド入力手順

2.3.3　バイナリモデル検査のメリット

これまでに述べたように、従来の形式的検証手法には様々なハードルがある。バイナリモデル検査はその問題の幾つかを解決するために考案した手法である。これから、どのような問題を解決したのかを説明していく。

言語間の記述能力の違いを解消

通常、あるプログラムを検査しようとした場合、その振舞いを捉えてモデル記述言語で記述し検査する。しかし、モデル記述言語には、ポインタ型、浮動小数点型、文字列型、多次元配列などが備わっていない場合が多い。PROMELA もそのような型を持っていない言語のひとつである。このように言語間に記述能力の違いがあると、プログラムの振舞いを厳密に表現出来ない。そこで、バイナリモデル検査では、同時に両方の言語を使用する。それにより、互いの言語の利点を生かしたモデル記述が可能となり、適切にプログラムの振舞いを捉えることが出来る。本研究では、SPIN の C 言語埋込み機能を利用して C 言語と PROMELA の同時利用を可能にしている。

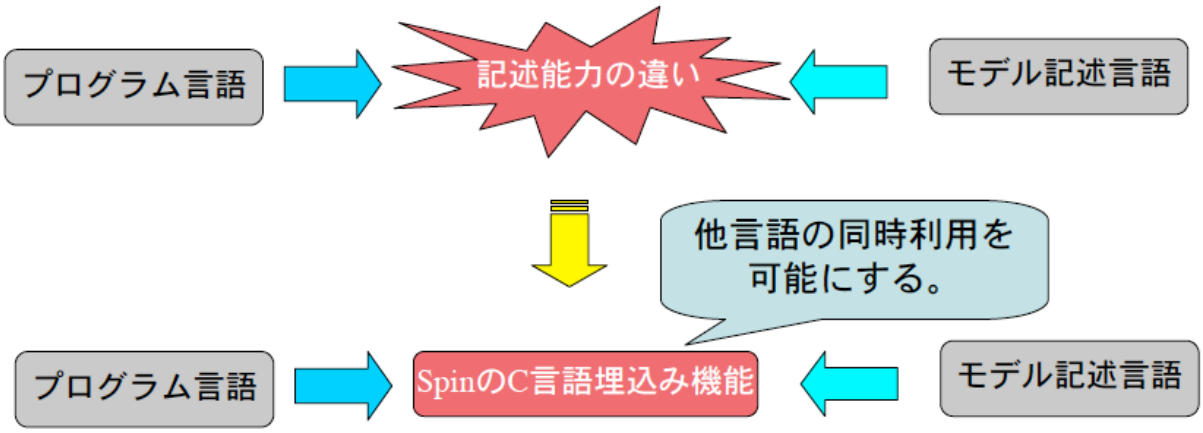

図 2.9: 言語間の記述能力の違いを解消

自動的に作成される検査モデル

モデル検査をする際、検査対象の振舞いを正確に記述した完成した検査モデルを作成する必要がある。しかし、検査対象をどの視点から見るかによってモデルは大きく変わり、正確なモデル作成は難しい。そこで、バイナリモデル検査のメモリ監視能力に着目した。監視するよう指定したメモリ空間を状態空間と見なす事により自動的にモデルを構築することが可能となる。指定するメモリ空間は実際に検査対象プログラムが動作する領域で

あり、検査用プログラムはメモリ操作を検査対象プログラムに任せ、その変化だけに注目して遷移システムを構築する。

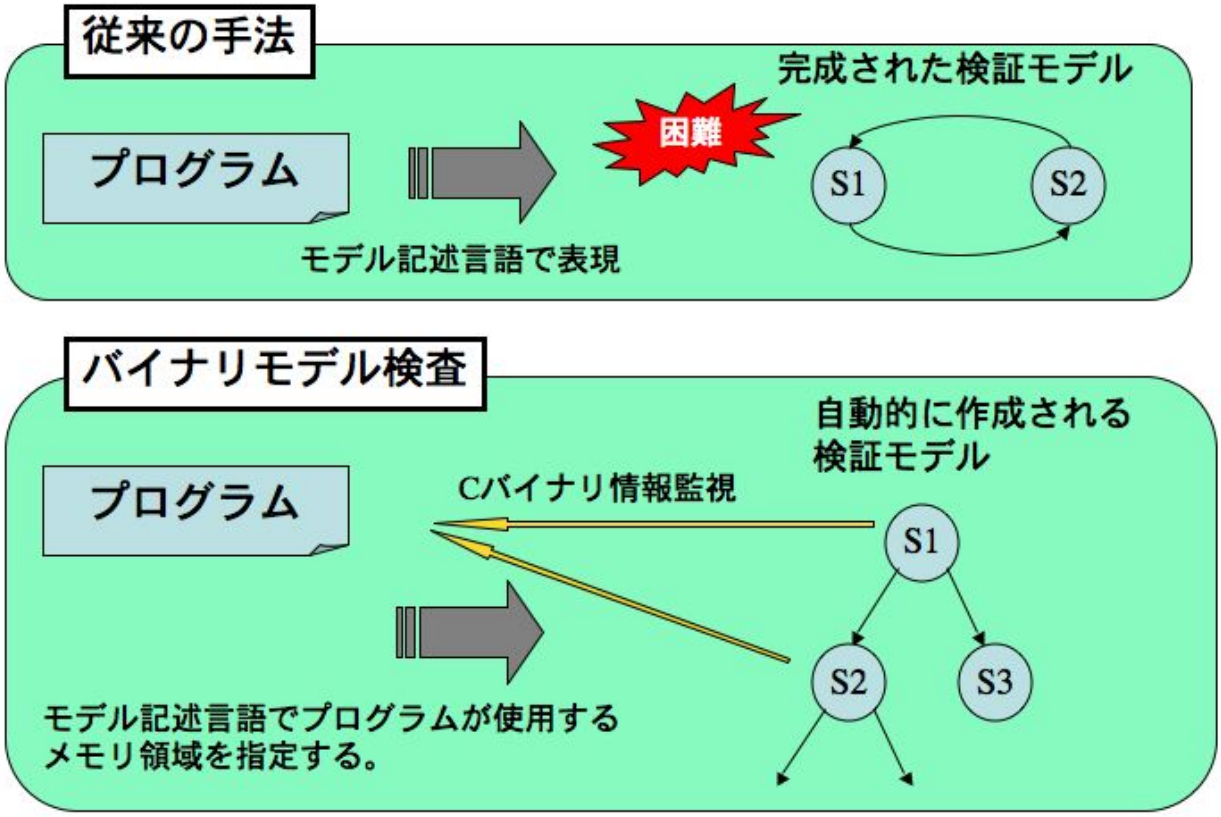

図 2.10: 自動的に遷移システムを構築する

既存の環境を利用する

従来の手法では、検査対象となる環境全てをモデル化する必要があった。例えば、組込みシステムを検査しようとした場合は、アプリケーションの他にもデバイスドライバやデバイスの振舞いをモデル化して検査を行う必要性があった。しかし、バイナリモデル検査では既存のプログラムをそのまま利用出来る機能がある。アプリケーションの部分だけをモデル化し、デバイスドライバ、デバイスは実機で動かすという手順が取れる。つまり、検査を行いたい箇所だけをモデル化し、あとは既存のものを直接利用、検査モデルは実機の結果を利用して遷移する。

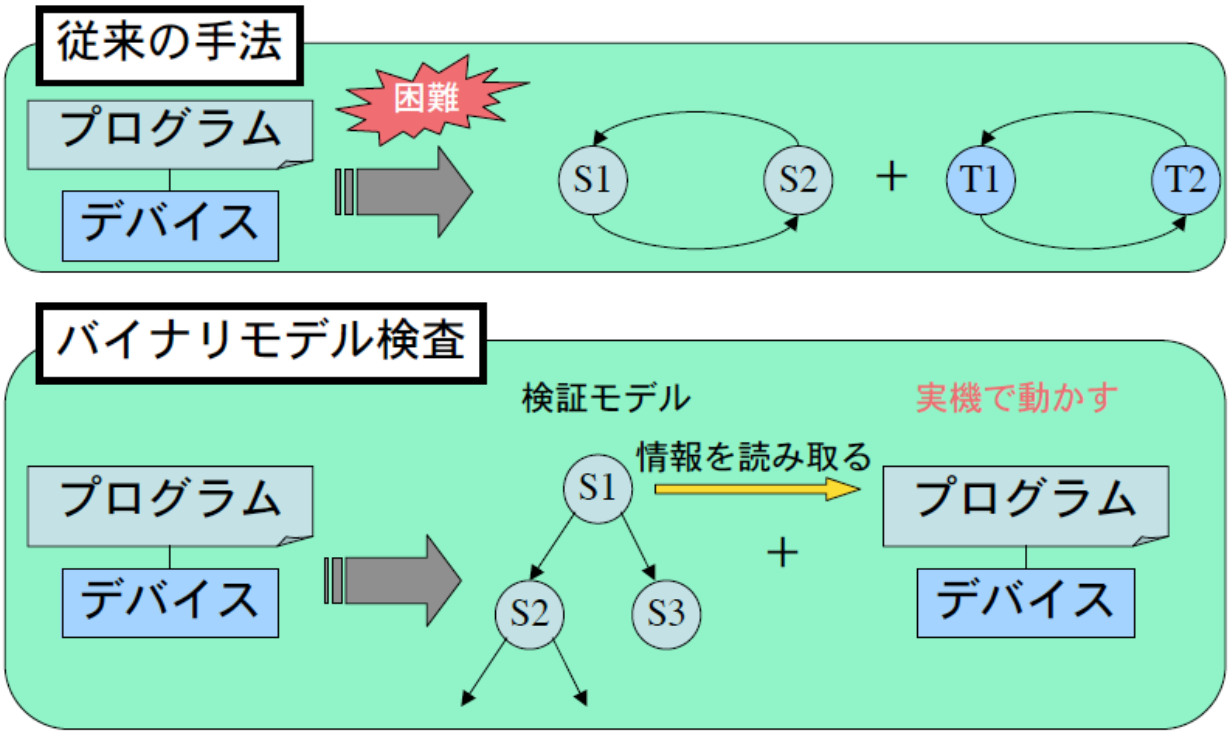

図 2.11: 既存の環境を利用する

反例の利用

モデル検査器は、表明において指定した性質が成立しなくなると誤った実行結果に至るまでの反例を示す。一般的なテスト手法では再現性の低いエラーに対しても、網羅的に探索するモデル検査器は何度実行しても正確に反例を出力する。表明に何を記述するのかという点については、調べる性質によって様々であり、難しい問題ではある。しかし、自分が設定した範囲において確実にエラーを発見していけるため、デバッグにおいて大きな手助けとなると考えられる。

事前条件チェック

埋込 C コードフラグメントの例 2 でも取り上げたが、c_code の事前条件チェックを利用して、配列の境界条件や NULL ポインタのチェックが可能である。入力される全ての値に対してあり得る全ての状態を容易に網羅的に検査できる。また、Modex[HS](4.5 節）という事前条件検査のための埋込 C コードを自動で作成可能なツールがあるので参考にしてもらいたい。注意したいのは、この機能は容易に事前条件チェックができるのことが魅力なのであり、詳細なレベルのチェックを目的としていない。もし、詳細なレベルまでチェックしたいのならば静的検査ツールを用いて行ってもらいたい。

2.3.4 バイナリモデル検査をソートプログラムに適用する

検査プログラムを図 2.12 に示す。この例では、入力値を生成する外部モデルを 1 つ作り、ソート対象の配列に任意の値を代入している。外部モデルを作った理由は、入力値がなければソートプログラムは動作せず遷移システムを構築することが出来ないからである。また、PROMELA 記述の非決定的遷移を利用すれば、様々な入力値の組み合わせを容易に生成できるため C 言語ではなく PROMELA で作成した。図 2.12 の byte 型変数 r で任意の値を代入している箇所が外部モデルに対応する。15 行目では、配列に数値入力の際に配列の境界チェックをしている。もし、配列外に値を入力しようとすれば、ここで検査が停止し反例を出力する。そして、任意の値をソートした結果をチェックしソートプログラムが正しく動作したかを調べる。図 3 の c_expr{check()} 部分がそれに対応する。ここで、PROMELA ではなく C 関数で結果をチェックしているのは、状態数の観点から C 関数を利用した方が有利だからである。図 2.13 からも分かるように、状態数やメモリ使用量が少ない。バイナリモデル検査を使用せずに全てを PROMELA で記述したモデル検査を行った場合、配列要素の入れ替え作業や一時変数の変化まで状態と見なされてしまうので状態空間はさらに大きくなると考えられる。

図 2.12 を見ると分かるが新たに作成した箇所は、値を作成する外部モデルと check 関数のみである。ソートプログラム自体には手を加えてはおらず、検査対象プログラムに対しての変更はほぼ行わなかったと言える。バイナリモデル検査の特徴である「既存プログラムの直接利用」を行うことが可能であり、検査モデルを容易に作成できることが分かる。また、図 2.13 から状態数とメモリ使用量に関しても少なく済んでいると考えられる。一般にソートプログラムをモデル化し検査する場合、ソート作業中の一時変数や配列の入れ替え作業による値の変化の全てが違う状態と判断されるのでより多くの資源を使用する事になる。

- この検査で行った事のまとめ
  - 要素数 MAX の配列に任意の要素を格納したとき、ソートプログラムが正しく動作するのかを検査する。
  - 同時に配列の BoundaryCheck を行う。
  - エラーが発生すれば反例が出力されるので解析を行う。
  - 検査のために作成したのは、入力値を生成するモデルと check 関数のみ。
  - 状態数やメモリ使用量が少ない。

```
 1 #define MAX 5
 2
 3 c_decl{#include "sort.h"}
 4
 5 c_code{#include "sort.c"}
 6
 7 active proctype test(){
 8    byte i=0, r=0;
 9    do
10    :: i < MAX ->
11       do
12       :: r++
13       :: break
14       od;
15       c_code[0 <= Ptest->i && Ptest->i < MAX]
16       {num[Ptest->i] = Ptest->r;};
17       i++
18    :: else -> break
19    od;
20    c_code{ sort();};
21    if
22    :: d_step{c_expr{check()} ->
23       c_code{print_status();};
24       assert(false)}
25    :: else
26    fi
27 }
```

図 2.12: sort.spin

```
State-vector 12 byte, depth reached 1299, errors: 0
     5890 states, stored
     1280 states, matched
     7170 transitions (= stored+matched)
        0 atomic steps
     hash conflicts: 10 (resolved)

    2.724   memory usage (Mbyte)
```

図 2.13: ソートプログラムの検査結果

第3章　バイナリモデル検査による検査

２章において、バイナリモデル検査の概要と効果を簡単に説明した。本章の実験では、ソートプログラムに比べて複雑な状態遷移を持つと考えられる ls プログラムを検査することでバイナリモデル検査手法の有効性と問題点について考察する。対象とするのは、ファイル情報をリスト化し出力する、自作の ls プログラムである。ls プログラムを選択した理由は、ソート、リスト構造などの基本的なアルゴリズムを含んでおり、検査として適していると判断したからである。

3.1　検査概要

ls プログラムへのバイナリモデル検査適用の前に検査法の概要を説明していきたい。本研究では、バイナリモデル検査の手順を図のように定義している。

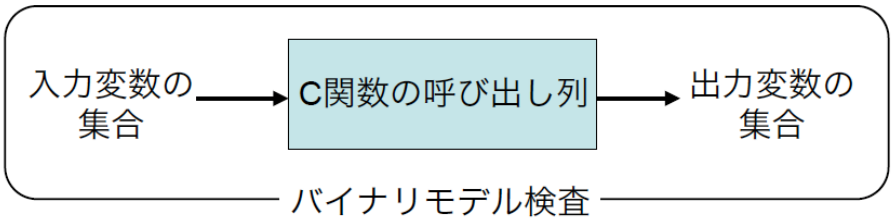

図 3.1: 検査の前提

- 入力変数の集合：一般的にプログラムには、動作させるために何らかの入力値が必要である。これは、それらプログラムへの入力値の集合であると定義する。実際には、関心のある入力値の組み合わせを Promela の非決定性を利用して作成し、入力変数の集合とする。
- C 関数の呼び出し列：同じ入力値であれば、C 関数の呼び出し列は同じであると定義する。また、そのような C 関数の呼び出し列になるように検査プログラムを作成する。また、ここで呼び出す関数自体は正しく動作する (セグメンテーションフォルトなどのシステムエラーを発生しないという意味) ものと仮定する。

- 出力変数の集合：入力変数の集合に対応して、出力し得る値を出力変数の集合と定義する。関数の呼び出しが正常に終了すれば出力変数の集合も正しく得られるはずである。実際の検査では、この出力変数の集合に対して表明をかけ、調査したい性質（プログラムの仕様）がC プログラム実行後に満たされているかを調べる。

- 状態変数の指定：C 変数の値の変化を監視したい場合、状態変数として監視するようモデル検査器に宣言する。宣言されたC 変数はモデル検査器の内部状態ベクトルに組み込まれ、状態として扱われる。

では、2.3.4 項で例に挙げたソートプログラムに対し、これらがどのように設定されていたかを説明する。

- 入力変数の集合；ここでの入力値はPROMELA 変数 r である。r にはPROMELA の非決定性から任意の値が代入される。そしてc_code 内でr は参照され、C の配列 num に代入される。つまり、r の取り得る値が入力変数の集合となる。

- C 関数の呼び出し列：今回のソートプログラムの検査では、呼び出したソート関数が正しく配列をソートしているかにのみ焦点を当てている。従って、呼び出すC 関数はソート関数のみである。複雑なバイナリモデル検査プログラムになると、この関数呼び出し箇所の記述が長くなる。

- 出力変数の集合：検査プログラム内では、出力されている変数 (ソートされた配列 num) の集合は確認できないが、C 関数 check() によって正しくソートされているかが調べられている。出力変数の集合の中に満たしたい性質を満たしていない部分集合が存在すれば表明によってエラーを出力するように記述してある。

- 状態変数の指定：ソートプログラムの例では、C 変数を状態変数として指定していない。なぜならこの例では、入力変数の集合に対しての出力変数の集合にしか注目していないからである。ソート対象である配列を状態変数として指定することもできるが、配列の内容の微妙な変化に今は興味はなく、無駄な状態を増やさないためにも状態変数の指定は行わなかった。

```
#define MAX 5

c_decl{#include "sort.h"}

c_code{#include "sort.c"}

active proctype test(){
  byte i=0, r=0;
  do
  :: i < MAX ->
    do
    :: r++
    :: break
    od;
    c_code[0 <= Ptest->i && Ptest->i < MAX]
    {num[Ptest->i] = Ptest->r;};
    i++
  :: else -> break
  od;
  c_code{ sort();};
  if
  :: d_step{c_expr{check()} ->
    c_code{print_status();};
    assert(false)}
  :: else
  fi
}
```

Promelaに非決定性を利用し、rに値を代入。入力値としてc_code内で配列numにrを与えている。

ソートプログラムの検証では、呼び出すC間数列はsort関数のみ。

ソートの結果である配列(出力値)をチェック関数で調べる。

図 3.2: sort.spin と検査法概要の対応関係

3.2　ls プログラムの仕様・構成について

この節では、本章でのバイナリモデル検査対象である自作 ls プログラムについて説明する。

ls プログラムは、システムからファイル情報を取得し出力する UNIX/Linux コマンドの代表的なプログラムである。今回は、その ls プログラムの機能を簡易化した myls プログラムを作成し、実験対象とした。myls のソースコード行数は、６００行程度である。

ls プログラムの仕様

ディレクトリ内各ファイルの属性を出力する./myls コマンドを作成。出力するファイルの情報はオプションによって異なるが、出力可能な情報は以下の通り。

＊ inode 番号、パーミッション、リンク数、ユーザー ID、グループ ID、ファイルサイズ、最終更新日時、ファイル名

- 実行方法

```
#./myls [-options] /PathName/

ex. #./myls
    #./myls -l .
    #./myls /home/user_dir/
```

図 3.3: myls 実行方法

  - 引数（PathName）を指定しなかった場合は、カレントディレクトリ内のファイルを表示する。引数を指定した場合は、指定したディレクトリ内のファイルを表示する。オプションについては次項で説明するが、オプションに基づいて表示情報やソートなどを行う。

- オプションについて

  - オプションは全部で４種類。
    * -l: 指定したディレクトリ内にあるファイルの全ての情報を表示する。
    * -a: 指定したディレクトリ内の全てのファイル (. や.. など) を表示する。
    * -t: ファイルの最終更新時間でソートする。
    * -r: 逆順でソートする。

ソートについては、デフォルトではファイル名でソートするようになっている。

ls プログラムの構成

ls プログラムを構成する主要な要素は図 3.4 の四角で囲った部分のようになっている。関数の仕様など詳細は付録を参照のこと。

- ディレクトリとオプションの解析：ユーザにより指定されたオプションの種類・組み合わせ、ls 表示するディレクリの場所などを解析する関数が存在する。ここで、ユーザが入力した文字情報を扱いやすいよう変換し、次の段階に渡す。

- システムコールによるファイル情報の取得：解析した情報をもとにシステムコールを呼び出し、参照したいファイルの情報を取得する。作業自体は、特殊な事をしているわけではない。しかし、この箇所が検査を困難にする要因の 1 つになっている。バイナリモデル検査中でシステムコールを呼び出す事自体は可能であるが、システムコールの挙動は ls プログラムだけでは決定しない。この事が検査を困難にしている要因である。システムコールの挙動は、他のタスクや OS の状態などに左右されてしまい厳密に ls のみの振る舞いを検査するのは難しくしている。詳細は、3.4 節において述べる。

- ファイル情報を動的に確保した領域に複製：ls プログラム中では calloc を呼び出し、取得したファイル情報のサイズ分だけ動的にメモリを確保している。ここにも、バイナリモデル検査特有の問題がいくつかある。1 つ目は、動的に確保したメモリは通常 Spin の内部状態ベクトルに追加して監視できないことである。2 つ目は、malloc-free の対応関係の問題である。モデル検査の探索木上において任意のタイミングで malloc した領域を free するとバックトラックできないという問題が発生する。これらの問題点については、3.8 節において述べる。

- オプションに従ってファイル情報 (複製) を並び替える：解析したオプションの情報に従って、前段階で複製したファイル情報を表示し易いよう並び替える作業を行う。実際には、ソート関数を呼び出しファイル名や最終更新時間などをもとに配列の並び替えを行う。

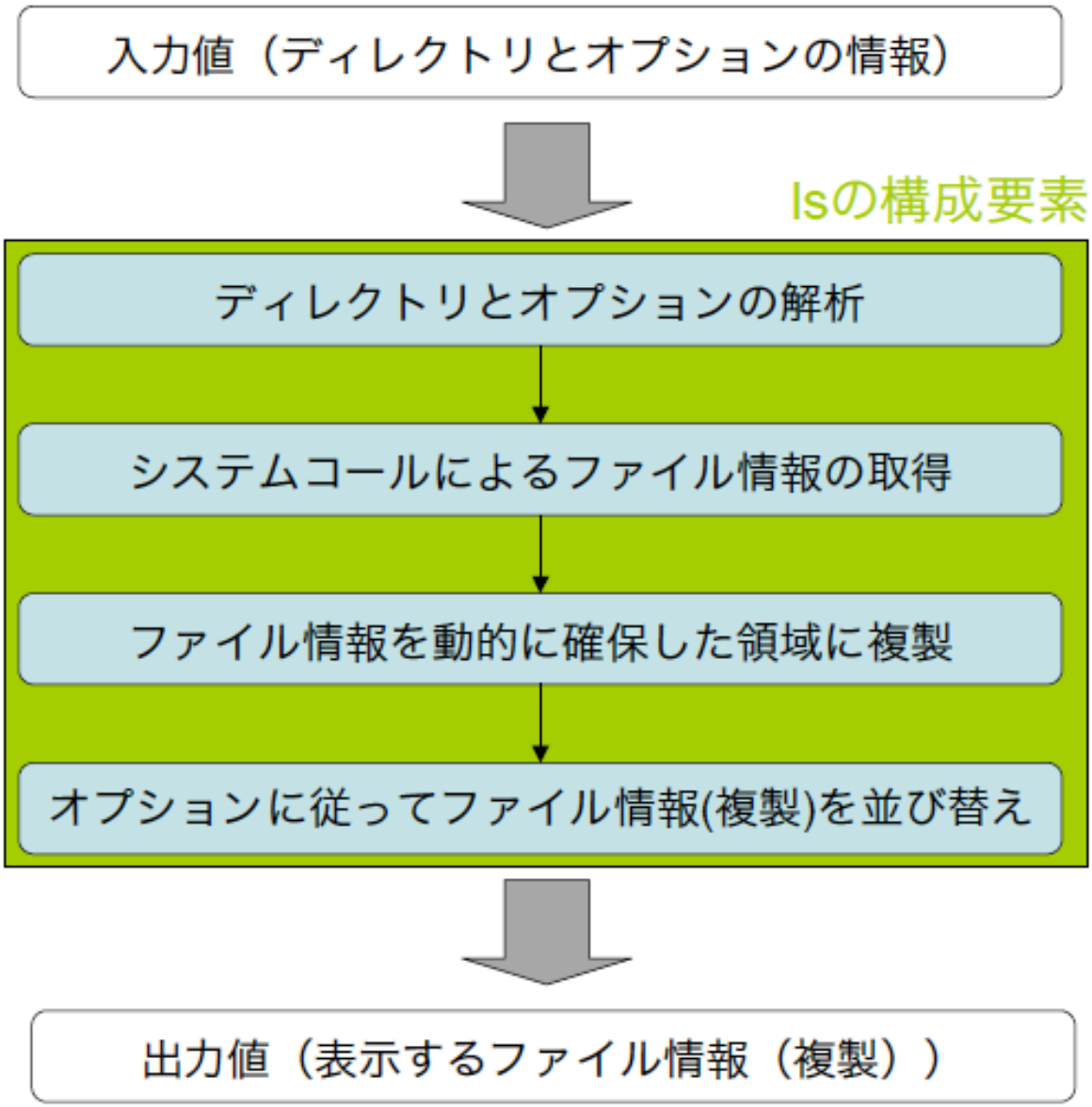

図 3.4: ls プログラムの構成

3.3　ls プログラム検査内容

これまでの説明においてバイナリモデル検査をする際には、C プログラムに対して何らかの形で入力変数の集合を与えなければならないことを述べた。そこで本節では、3.1 節での検査概要、3.2 節での ls プログラムの仕様・構成に基づいて、ls プログラムにはどのような入力変数の集合を与えたのかについて説明する。ls プログラムに与えた値は以下の２つである。

1. オプションの組み合わせ：3.2 節の仕様に基づいてコマンド入力され得る全てのオプションの組み合わせを入力変数の集合として与える。ただし、入力変数として与える値の中に、存在しないオプションを含めたりはせず、４つのオプションの組み合わせのみとする。したがって、オプションの最長は４である。また、同じオプションが４つ続く場合も正しい入力として考える。

2. 読み込むファイル数：図 3.4 の構成からも分かるように ls プログラムは、ファイル情報をシステムコールを呼び出すことで獲得し処理を行う。そこで、入力変数の集合として読み込むファイル数を０～１万まで変化させて ls プログラムに与える。ファイル数はユーザが与える入力変数ではないが、ls プログラムにとっては外部から得られる情報であるので本研究では、入力変数としてファイル数を捉えた。

次節において２つの検査環境の設定を行うが、１つ目の環境に対してはオプションの組み合わせを入力変数の集合として与え、２つ目の環境に対しては、読み込むファイル数を入力変数の集合として与えた。実際に ls プログラムを検査する際にどのようにして入力変数の集合を与えたのかについても次節を参考にして欲しい。

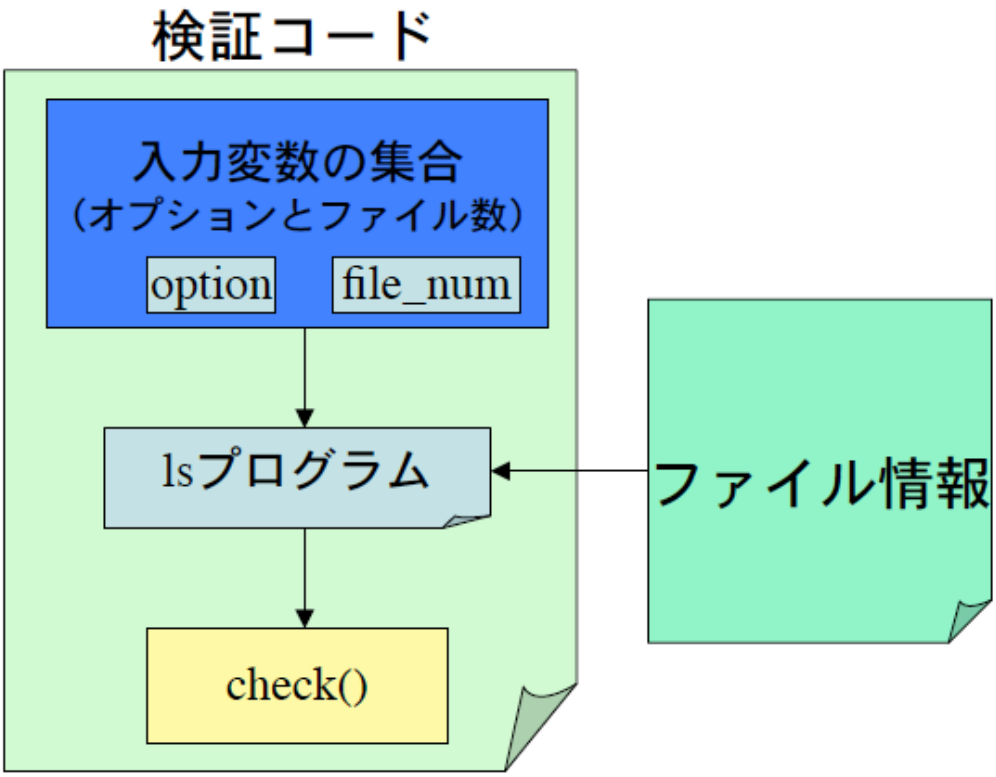

図 3.5: ls プログラムの検査内容のイメージ

3.4　検査環境の設定

3.1 節において検査方法を定義し、ソート関数の検査例を示したが、ls をバイナリモデル検査する場合にはどのように検査環境を設定するのかについて本節ではより詳細に述べる。検査環境を設定する場合、大きく分けて２つの方針がありどちらかを選択する必要がある。これは、前述の ls プログラムの構成で簡単に述べたが、システムコールの呼び出しに原因がある。なぜシステムコールに原因があるのか、方針を選択するかによって検査で保証できること、できないことがどのように異なってくるのかを説明していく。以下の項目は、２つの方針に関しての説明である。

- 検査環境をそのまま使う場合
  - モデル検査を行う場合、検査環境となる領域をモデルという定まった形で定義し、検査を行う。しかし、バイナリモデル検査の場合、動作環境を呼び出して、そのまま検査に利用することができる。このように検査する場合を『検査環境をそのまま使う場合』と呼ぶ事にする。
- 検査環境をエミュレートする場合
  - バイナリモデル検査では、上記のように検査環境をそのまま使って検査が行える。しかし、厳密に定義されたモデルの中で動作環境を含めて網羅的に検査を行うことも非常に重要な事である。よって、動作環境を抽象化し、厳密な定義を定め、検査環境とする。このような場合を以後、『検査環境をエミュレートする場合』と呼ぶ事にする。

3.4.1　検査環境をそのまま使う場合

概要

１つ目の方針は、検査対象の C プログラムからシステムが提供しているシステムコール（動作環境）を直接呼び出す検査方針である。この方針の利点は、既存のリソースをそのまま利用することで検査にかかる作業量が減る点やプログラム運用環境そのものの上での検査が可能になる点である。しかし、それ故の問題点もある。システムコールの反応は、システムの状態に左右されるのでどのように振る舞うのかをバイナリモデル検査上で定義できないのである。１回目の検査では出なかったエラーが全く同様の２回目の検査で出る可能性もシステムの状況によってはあり得る。よって、検査結果が OS や他タスクの状態に左右されてしまい、厳密に検査対象プログラムを検査ができない。しかし、安定しているシステム、検査中に他タスクとの競合が発生し難いと分かっている環境においては、既存のリソースを直接呼び出して実行しながら検査できる本方針は有益だと考えられる。

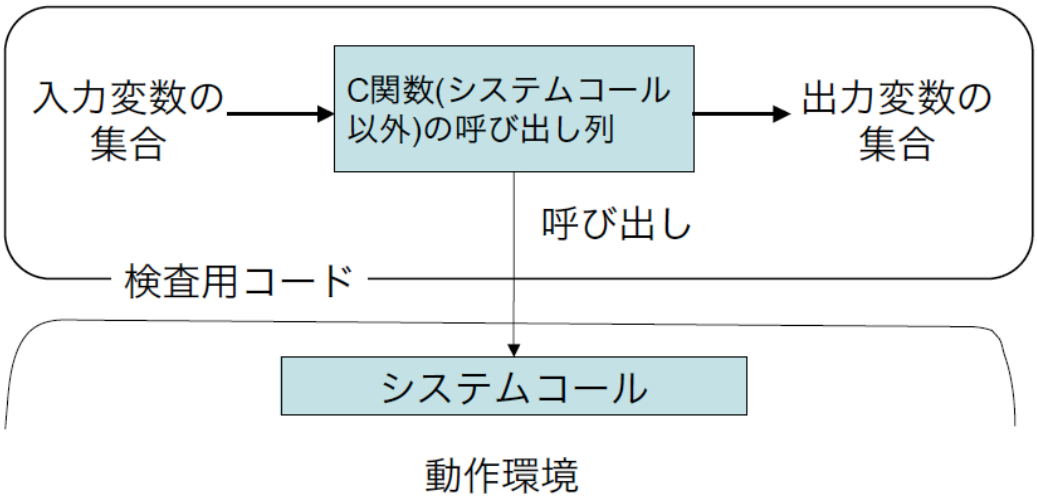

図 3.6: 検査環境をそのまま使う場合

手順

では、検査環境をそのまま使う場合の概要に基づいてどのような手順をとってバイナリモデル検査を行うかについて説明する。

1. 入力変数の集合を作成するモデルを作成する。複数の入力変数を扱う場合は、PROMELA で非決定的な記述をすればC言語で入力値を与えるよりも容易に網羅的な値を作成することができる。

2. プログラムを構成する関数列を c_code を使用して呼び出す。この方針は、検査環境をそのまま使う場合なので関数列にシステムコールが含まれていてもそのまま呼び出せば良い。

3. 出力変数の集合が保証したい性質を満たしているのかを調べる。

ls プログラム検査での実装手順

前述した手順に従って ls プログラムの検査ではどのように実装したかについて説明する。

1. 3.3 節において ls プログラムの検査内容について説明したが、検査環境をそのまま使う場合において今回は任意のオプションの組み合わせを入力変数の集合とした。図 3.7 と図 3.8 が実際に ls プログラム検査コード内でオプションの組み合わせを生成している PROMELA コードである。この 2 つで何が違うかというと入力されるオプションの組み合わせのみを生成している場合が図 3.7 であり、順列を生成してい

るのが図 3.8 である。オプションは litr の 4 つであるが、この 4 つの組み合わせにのみ興味がある場合は図 3.7 のようにコードを記述すればよい。そのようにすれば状態数も抑えられる。また、オプションの順番や例えば rrrr のように同じオプションが 4 つ続くといった入力値まで考慮したいのならば図 3.8 の用に記述すれば良い。補足だが opt1,opt2,opt3,opt4 はオプションの入れ物であり、” litr の 4 つのオプション ” と ” オプションなし ” から任意の値が選ばれ代入され、オプションの順列が生成される。このようにすることで状態数は多くなってしまうが、現実に入力されうるオプション列により近い入力変数の集合を生成することが可能となる。

```
 1   if
 2     ::c_code{addStr("l");};__l=1;
 3     ::skip;
 4   fi;
 5   if
 6     ::c_code{addStr("i");};__i=1;
 7     ::skip;
 8   fi;
 9   if
10     ::c_code{addStr("t");};__t=1;
11     ::skip;
12   fi;
13   if
14     ::c_code{addStr("r");};__r=1;
15     ::skip;
16   fi;
17
18   if
19     ::c_expr{strcmp(myargv[1],"-") == 0}
20        ->d_step{c_code{ myargc = 1; myargv[1]="~/";};};
21     ::skip;
22   fi;
```

図 3.7: 検査用コードにおいてオプションの組み合わせを生成する箇所 (オプション順番に興味なし)

```
 1  if
 2     ::opt1=T;
 3     ::opt1=L;
 4     ::opt1=I;
 5     ::opt1=R;
 6     ::opt1=NON;
 7   fi;
 8   if
 9     ::(opt1==NON)->goto L1;
10     ::else ->
11        if
12  ::opt2=T;
13  ::opt2=L;
14  ::opt2=I;
15  ::opt2=R;
16  ::opt2=NON;
17        fi;
18   fi;
19   if
20     ::(opt2==NON)->goto L1;
21     ::else ->
22        if
23  ::opt3=T;
24  ::opt3=L;
25  ::opt3=I;
26  ::opt3=R;
27  ::opt3=NON;
28        fi;
29   fi;
30   if
31     ::(opt3==NON)->goto L1;
32     ::else ->
33        if
34  ::opt4=T;
35  ::opt4=L;
36  ::opt4=I;
37  ::opt4=R;
38  ::opt4=NON;
39        fi;
40   fi;
```

図 3.8: 検査用コードにおいてオプションの組み合わせを生成する箇所 (オプション順番に興味あり)

2. ls プログラムを構成する関数列を c_code を使用して呼び出す。今回の検査では、ls プログラムの main 関数内で呼ばれている順で関数列を呼び出した。

```
 1  L1:
 2    c_code{makeOption(Ptest->opt1,Ptest->opt2,Ptest->opt3,
                                                  Ptest->opt4);};
 3    c_code{printf("myargv %s",myargv[1]);};
 4    c_code{Ptest->dir = chkFlags(&(Ptest->flag),
                                          myargc,myargv);};
 5    c_code{Ptest->start_dir = getenv("PWD");};
 6
 7    c_code{Ptest->file_num = cnt_dir(Ptest->dir);};
 8
 9    c_code{Ptest->result = check_file_num(Ptest->file_num);};
10
11    do
12      ::c_expr{Ptest->result == -1} ->
13                  c_code{printf("Error cnt_dir function");};
                    assert(false)
14      ::c_expr{Ptest->result == 0} ->
15                  c_code{printf("No File");};assert(false)
16      ::else -> break
17    od;
18
19    c_code{chdir(Ptest->start_dir);};
20    c_code{Ptest->list = mycalloc(Ptest->file_num);};
21    c_code{Ptest->seek = seek_dir(Ptest->dir,Ptest->list,
                                        Ptest->file_num);};
22
23    do
24      ::c_expr{Ptest->seek == -1} ->
25          c_code{printf("Fail seek_dir");};assert(false)
26      ::else -> break
27    od;
28
29    c_code{mysort(Ptest->list,Ptest->file_num,sizeof(Flist),
                                    f_comp,&(Ptest->flag));};
```

図 3.9: 検査用コードの C 関数呼び出し列部分

図 3.9 と ls プログラムと比較をすると分かるのだが、一部記述は異なるが、ほぼ ls プログラムの main 関数内の通りに関数を呼び出している。

3. 出力変数の集合が保証したい性質を満たしているのかを調べるため検査用コードでは以下のようにしている。

```
1  c_code{Ptest->sort =
                    sortCheck(Ptest->list,Ptest->file_num);};
2   if
3     ::d_step{c_expr{ Ptest->sort < 0} ->
4         c_code{printf("check = %d",check);}-> assert(false)}
5     ::else -> break
6   if;
```

図 3.10: 検査用コードの check 関数部分

表示するファイル情報が格納されている変数値を sortCheck という C のチェック関数に渡している。sortCheck は渡されたファイル情報が意図された通りにソートされているかを調べる関数である。3 行目ではその返り値を条件文にかけ、エラーがあれば表明によって反例を出すようにしている。また、このようにプログラムの仕様に基づいてチェックを行う関数のバリエーションを増やせば調べることができる性質も増やす事ができる。

3.4.2 検査環境をエミュレートする場合

概要

2 つ目の方針は、検査環境の C プログラムが呼び出す動作環境をモデル化し、エミュレートしながら検査する方針である。この方針の利点は、プログラムの振る舞いを動作環境を含め、網羅的に検査できることである。検査環境をそのまま使う場合とは異なり、システムの状態に左右されずに検査を行う事ができる。よって入力変数の集合が同じ場合は、何度検査プログラムを実行させても同じ検査結果が必ず得られるのである。しかし、問題点もある。検査環境をそのまま使う場合とは逆に、システムが提供している箇所もモデル化する必要が出てくるので検査に必要なコストが大きくなってしまう点である。また、動作環境の振る舞いを抽象化して検査を行うため、検査によって保証できる性質を見定める必要も出てくる。詳細に動作環境の振る舞いをモデル化して保証できる性質の範囲を広げることも可能だが、コストがかかる上に状態爆発も発生しやすくなり、あまり現

実的ではない。しかし、保証したい性質が検査実行の前段階で見定められている場合は、厳密に性質を保証する事が可能であるのでこの方針が適していると考えられる。

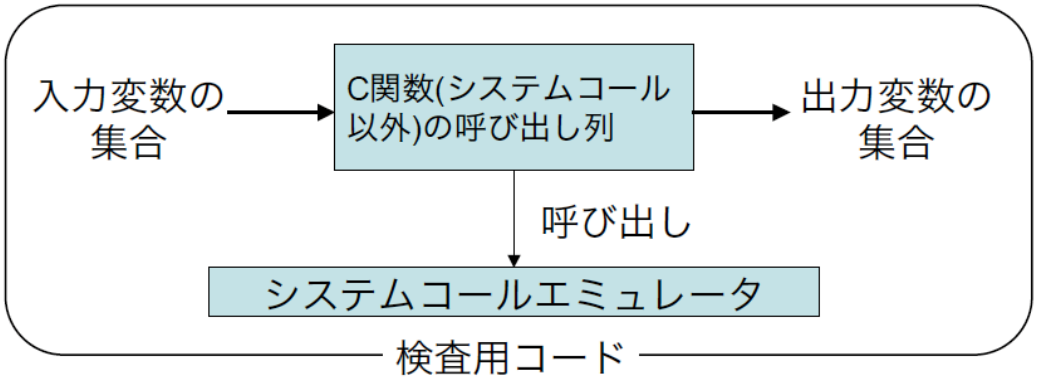

図 3.11: 検査環境をエミュレートする場合

手順

では、検査環境をエミュレートする場合の概要に基づいてどのような手順をとってバイナリモデル検査を行うかについて説明する。

1. 入力変数の集合を生成するモデルを作成する。複数の入力変数を扱う場合は、PROMELA で非決定的な記述をすればC言語で入力値を与えるよりも容易に網羅的な値を作成することができる。

2. 呼び出す関数列から動作環境を呼び出す箇所があった場合、その動作環境をエミュレートするC関数を自作して本来のものと置き換える。どの程度の精度までエミュレートする環境を作成するのかは検査したい性質によって異なるのでここでは言及しない。

3. 出力変数の集合が保証したい性質を満たしているのかを調べる。

ls プログラム検査での実装手順

前述した手順に従ってls プログラムの検査ではどのように実装したかについて説明する。

1. 3.3節においてls プログラムの検査内容について説明したが、検査環境をエミュレートする場合において、今回は読み込むファイル数を入力変数の集合とした。図3.12は、ls プログラム検査用コードの一部を抜粋したものである。

```
 1 do
 2     ::c_expr{Ptest->array <= 10002}->
 3                       c_code{array=Ptest->array;Ptest->array++;};
 4                       /* for initializing external model */
 5                       c_code{initStructStat(files);};
 6                       c_code{initDirInfo(dirInfo);};
 7                       .
 8                       .
 9                       .
10                       .
11     ::c_expr{Ptest->array > 10002} -> break
12   od
```

図 3.12: ファイル情報生成部分

図3.12のコードは、抜粋したものであるので分かり難いかもしれないが、Ptest→arrayがファイル数を保持する変数である。この変数を０～１万まで変化させることによって読み込むファイル数を変化させている。ただ、ここで問題がある。どのようにして読み込むファイル数を変化させるかである。考えられる方法は２つある。１つ目は、実際のシステム上にあるファイルを０～１万まで変化させてls プログラムの動作を調べるという方法である。２つ目は、ls プログラムが読み込もうとするファイル情報のをシステムのものではなく、自作した外部モデルにしてしまうというものである。結論から言えば後者を選択することにした。１つ目の案を実現するために、システム上に不用意に大量のファイルを作成するのはシステム上良くないと判断したからである。また、忘れてならないのが今回行っている検査が検査環境をエミュレートする場合の検査だということである。実際のシステム上にファイルを作ってしまうとどうしてもシステム依存の箇所が出てきてしまうので、システムと検査を切り分けることが難しくなってしまう。そこで、今回はファイル情報を仮想的に持つような外部モデルをC言語で作成し検査用コードの中から読み込む事とした。こうすることでシステムと検査プログラムの切り分けが容易になる。C言語でファイル情報モデルを作成した理由は、これも状態数をなるべく抑えるためである。図3.12の５、６行目の関数がそのファイル情報モデル初期化のための関数である。また、図3.13は、図3.12で呼び出している２つの初期化関数の定義である。指定されたファイル数Ptest→array分だけファイル情報の初期化を行っている。実は、初期化している構造体はあらかじめ用意した要素数１００００２(ファイル数の最大値が１万＋２であるため。＋２は”.” ”..”を表すために作成。)の静的な構造体配列である。実質的には、この配列がファイル数に応じて値を変えファイル情報を保持する外部モデルとなる。また、２つ目の関数initDirInfoは、ディレクトリのファイル情報を

保持する mydirent 構造体を初期化する関数である。

```
 1 void initStructStat(struct stat* tmp){
 2   int i;
 3   time_t now;
 4   ino_t inode = 0;
 5   off_t off = 0;
 6   time(&now);
 7
 8   for(i=0; i< array; i++){
 9
10     tmp[i].st_mode = S_IFREG;
11     tmp[i].st_ino = inode++;
12     tmp[i].st_nlink = 1;
13     tmp[i].st_uid = 501;
14     tmp[i].st_gid = 501;
15     tmp[i].st_size = off++;
16     tmp[i].st_atime = now;
17     tmp[i].st_mtime = now;
18    }
19 }//end of initStructStat()
20
21 void initDirInfo(mydirent* tmp){
22   int i;
23   char name[256];
24
25   for(i=0; i< array; i++){
26     if(i==0)
27       sprintf(tmp[i].d_name,".");
28     else if(i==1)
29       sprintf(tmp[i].d_name,"..");
30     else
31       sprintf(tmp[i].d_name,"%d",i-2);
32   }
33 }//end of initDirInfo()
```

図 3.13: ファイル情報作成のための外部モデル

2. ls プログラムはもともとシステムのファイル情報を読み込むプログラムである。従って、内部では複数のシステムコールを呼び出している。そのため、検査環境をエミュレートするために、概要に準じてシステムコールをエミュレートする自作関数と置

き換える必要がある。図 3.14 は、システムコールをエミュレートする関数の定義を一部抜粋したものである。

myreaddir 関数は、システムコール readdir をエミュレートしたものである。myreaddir 関数は、通常システムコール readdir が読み込むはずの dirent 構造体ではなく、手順 1 で用意した mydirent 構造体の内容を読み込む。そのようにエミュレートすることで ls プログラム上で readdir → myreaddir と置き換えるだけで ls プログラムの他の箇所を変更することなく検査を行う事ができる。他のエミュレートしている関数も同様だ。mychdir 関数は、システムコール chdir をエミュレートしたものであり、mystat 関数は stat 関数をエミュレートしたものである。特に、mychdir 関数に至っては本当にカレントディレクトリを変更する必要がないので空の関数である。また、今回のエミュレートでは、システムコールの返すであろうエラー値までは考慮に入れていない。もし、エラー処理も含めた検査がしたいならばシステムコールのエラーに関してもなんらかのモデル化が必要であると考えられる。

```
 1  /* dirent関数をエミュレート */
 2  struct mydirent* myreaddir(DIR* dp){
 3    static int i = 0;
 4    if(dp == NULL)
 5      i = 0;
 6    else{
 7      if(i < array){
 8        i++;
 9        return &(dirInfo[i-1]);
10      }
11      else{
12        i = 0;
13        return NULL;
14      }
15    }
16  }//end of myreaddir()
17
18  /* chdir関数をエミュレート */
19  void mychdir(char *dir){
20
21  }//end of mychdir()
22
23  /* stat関数をエミュレート  */
24  void mystat(char *d_name,struct stat* mystat){
25    int index;
26    index = atoi(d_name);
27
28    mystat->st_dev = files[index].st_dev;
29    mystat->st_ino = files[index].st_ino;
30    mystat->st_mode = files[index].st_mode;
31    mystat->st_nlink = files[index].st_nlink;
32    mystat->st_uid = files[index].st_uid;
33    mystat->st_gid = files[index].st_gid;
34    mystat->st_rdev = files[index].st_rdev;
35    mystat->st_size = files[index].st_size;
36    mystat->st_blksize = files[index].st_blksize;
37    mystat->st_blocks = files[index].st_blocks;
38    mystat->st_atime = files[index].st_atime;
39    mystat->st_mtime = files[index].st_mtime;
40    mystat->st_ctime = files[index].st_ctime;
41   }//end of mystat()
```

図 3.14: システムコールエミュレター

3. check 関数の部分に関しては、検査環境をそのまま使う場合と全く同じである。図 3.10 と異なっているのは sortCheck 関数の返り値を一旦保持してから c_expr で判定するか、直接 c_expr 内で判定するかの違いだけである。従って、行っている事の本質は同じであり、保証したい性質の内容によって check 関数のバリエーションを変更すればよい。

```
1  if
2       ::c_expr{sortCheck(Ptest->list,Ptest->file_num) < 0} ->
3          c_code{printf("check = %d\n",check);}-> assert(false);
4       ::else -> skip
5  fi;
```

図 3.15: 検査用コードの check 関数部分

3.5 監視する C 変数の指定法

Spin を使用してモデル検査を行う場合、Promela 言語で記述された変数は全て Spin の内部ステイトベクターに組み込まる。そして全変数が Spin によって自動的に監視される。しかし、埋込 C を利用したバイナリモデル検査の場合は、監視したいのは C 変数である。前述した c_state や c_track を用いて、明示的に監視したい C 変数を指定し、メモリ領域全体から C 変数が使用している一部の領域のみを抽象化によって取り出し、C 変数を状態として扱えるようにしなければならない。もし、検査環境がソートプログラムのような十数行であるようなプログラムならば全ての C 変数を指定してしまえば良いかもしれない。けれど、それ以上の規模のプログラムとなると監視する変数が増えてしまい、状態爆発の原因となってしまう。

そこでバイナリモデル検査では、図 3.16 のように興味がある特定の C 変数のみを指定し、Spin の内部ステイトベクターに加える。そのようにすることで検査に必要な状態空間を抑えることが可能となり、ある程度規模がある C プログラムの検査も可能となる。

ただし、特定の C 変数のみを指定することによって問題も発生する。以降の節でこの問題について取り上げていく。また、C ソースコード中で宣言した変数全てを埋込 C 機能を使って監視できるわけではない。この問題についても後述するが、今までの説明からも分かるように監視する C 変数を決定する作業は非常に難しく、形式化することは現段階では難しい。そこで、発生した個々の問題に対してどのように対処したかについて説明していく。

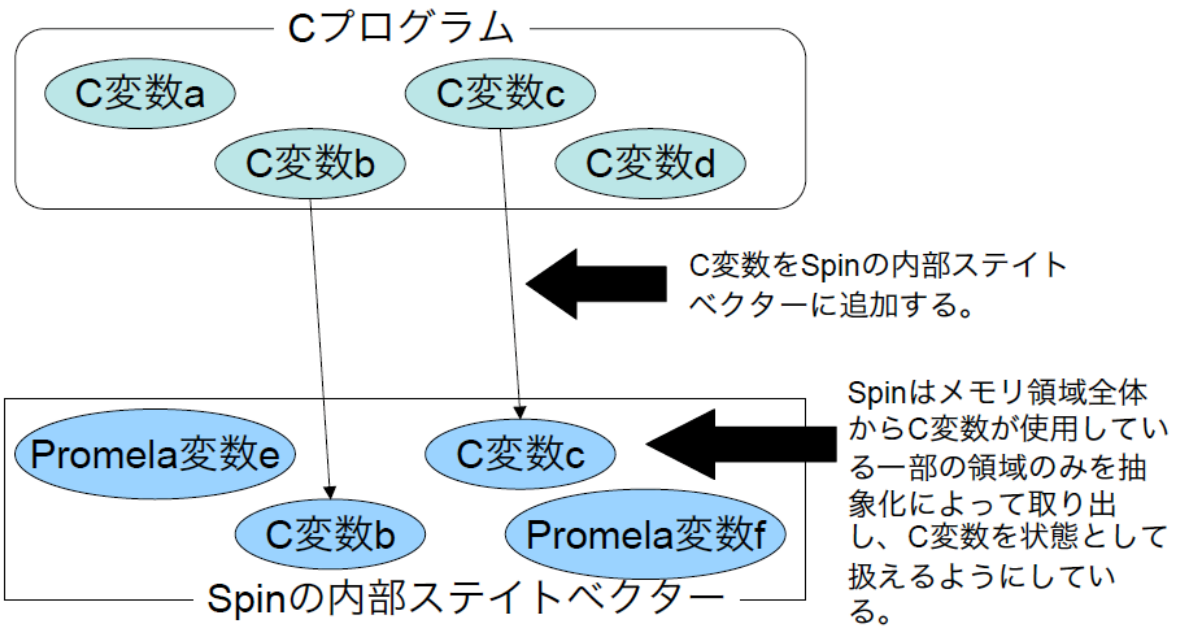

図 3.16: 特定の C 変数のみを指定する

（注意!!）上述の説明を読むと全ての C 変数が SPIN から監視可能であるかのように思える。しかし、SPIN の内部ステイトベクターの管理方法上の理由で検査用プログラムが開始する時点で静的に確保されている C 変数 (c_track,c_state で指定・作成できる変数) のみが監視可能ではある。別の言い方をすれば、検査用のプログラムが開始する時点で、監視したい C 変数の先頭アドレスが分かっていないと SPIN の内部ステイトベクターに状態情報管理用の領域を作成できないということである。もちろん、そうでない C 変数 (動的変数や局所変数) の監視方法についても議論するが、特に注意しない限り、以下の C 変数という言葉は C 大域変数を指すものと考えてほしい。

3.6 C 大域変数の取り扱い

3.5 節で説明したようにバイナリモデル検査において、状態と見なすために指定する C 変数は通常、全体の一部だけである。そのため、注意を払わなければならない事項がある。SPIN の内部ステイトベクターに追加しなかった C の大域変数の取り扱い方である。あるパターンに当てはまる使い方をされる C の大域変数に対して、適切な処理を行わなければならないのである。どのようなパターンに当てはまる C 変数なのかを説明していく。

3.6.1　どのようなC大域変数に注意すべきか

SPIN によって監視されていない C 大域変数になぜ注意が必要なのかというと、バイナリモデル検査を行うにあたり、C プログラムが何度も呼ばれるからである。普通 C プログラムが実行される場合は、値が入力され、関数列が呼び出され、値を出力し、終了する。しかし、バイナリモデル検査で同じ C プログラムを呼び出すと事情が異なってくる。3.1 節において検査法概要を説明したが、バイナリモデル検査では、C プログラムに入力変数の集合を与える。つまり、図 3.18 のように C プログラムに対する入力変数が変わりながら検査が行われていくのである。ここで、注意して欲しいのは、２回目の C プログラム呼び出しで使用する C 大域変数は、1 回目の C プログラム呼び出しで使用された状態のままになっているということである。もし、この C 大域変数が SPIN によって監視されている場合ならば自動的に、この C 大域変数は初期化される。しかし、監視されていない場合は２回目の C プログラム呼び出し前に、自分で初期化する必要性が生まれるのである。次項では、注意すべき C 大域変数についてもう少し一般化された観点から議論する。

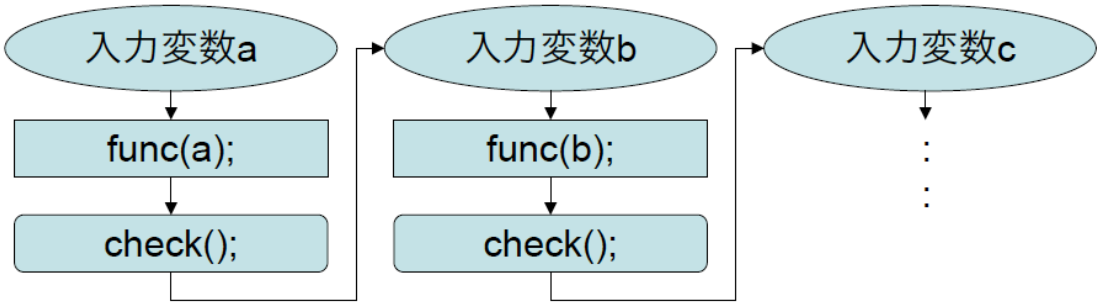

図 3.17: バイナリモデル検査の動作イメージ

3.6.2　注意すべきC大域変数に関する処理の一般化

ここまでで、なぜ C 大域変数に対して処理を行わなければならないかについて述べてきた。そこで、本項では処理を行わなければならない C 大域変数の判別方法、処理の仕方をより一般化して議論する。

図 3.18 は、SPIN がモデル検査をする際の探索木の一部を抜粋したものであると捉えて欲しい。C プログラム ref は、星印のポイントで C 大域変数を参照する。そして、そのパスの探索が終われば根に戻り、他のパスを非決定的に選択、選択したパスの C プログラム ref を呼び出し C 大域変数を参照する。しかし、図 3.18 のような探索木が生成される検査用コードを記述した場合、C 大域変数の振る舞いは正常なものではなくなり、バイナリモデル検査自体が失敗してしまう。なぜなら、このような探索木が生成される検証用コード

では、同じパスの同じポイントに置いて ref を実行しても探索の順によって C 大域変数の値が異なるからである。例えば、左の ref 後に中心の ref を呼び出した後の C 大域変数の値と、右の ref 後に中心の ref を呼び出した後の C 大域変数の値とが異なる現象がこの探索木では発生し得るということである。SPIN の内部ステイトベクターに C 大域変数を組み込まないことで、C 大域変数が直前の ref で使われた状態のまま次のパスに持ち越されてしまい、このような問題が発生する。さらには、そのような使い方をされている C 大域変数こそが処理すべき変数である。そこで、この問題の解決策を一般化した図が図 3.19 である。図 3.19 では、菱形のポイントで C 大域変数を初期化し、その後、星印のポイントで C 大域変数を参照している。つまり、

- 非決定的なパス選択の後かつ ref の前で C 大域変数を定数 orSPIN 内部ステイトベクターの値で初期化している。このようにすればパスの選択順によって C 大域変数の値がパスの同じポイントでは同じになる。

のである。ここで、初期化するための値である定数 or ステイトベクターについて説明する。定数に関しては考え易い。全ての init 箇所において同じ値で初期化してしまうというやり方である。このようにすれば、上述の通りどの順でパスを選んでも初期値は同じであるので必ず同一ポイントでの ref の値は同じになる。同様に、初期値の値が SPIN の内部ステイトベクターに組み込まれている値であってもよい。なぜならどのような順で探索パスが選ばれたとしても SPIN が自動的にそのパスに応じた初期値を設定してくれるからである。よってこの場合も上述の通り、どの順でパスを選んでも初期値は同じであるので必ず同一ポイントでの ref の値は同じになる。

次項では、この一般化をもとに、簡単な例で実際にバイナリモデル検査を行った例を示す。

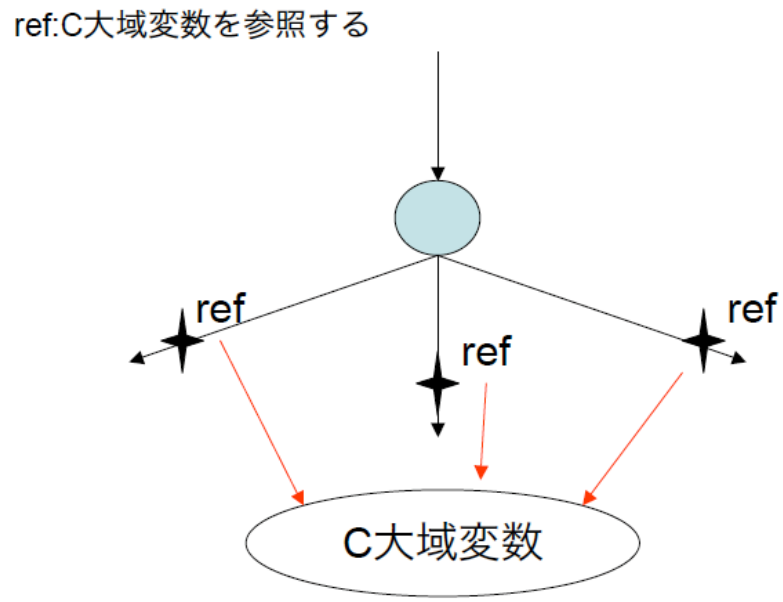

図 3.18: C 大域変数の振る舞いがおかしくなる例

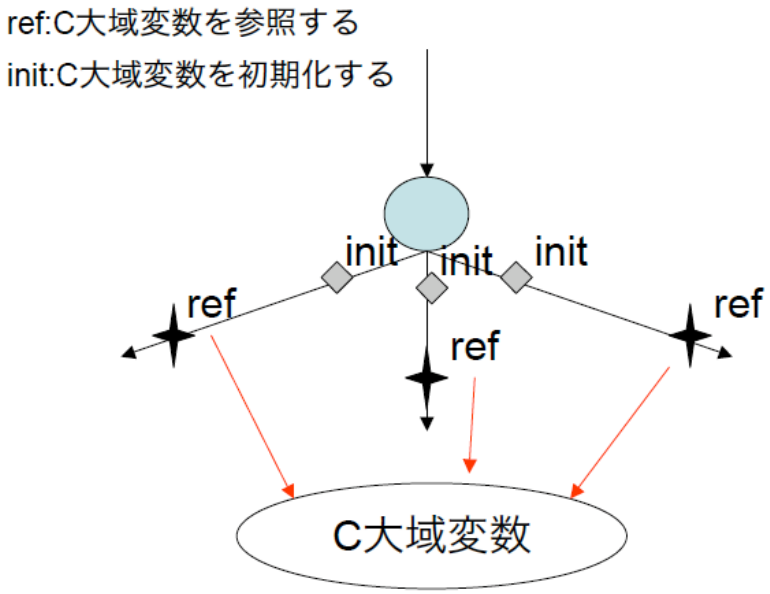

図 3.19: C 大域変数が正常に振る舞う例

3.6.3 初期化が必要な大域変数の例

図 3.20 と図 3.21 は、どちらも calcMax 関数が正しく動作するのかを入力変数の集合を与えることで検査しようとしたバイナリモデル検査の例である。前述した注意すべき C 大域変数 (int 型変数 max_num) がソースコード上に存在するため、検査が正常に行われるよう max_num の処理を行っている。どちらの図の検査も正しく行われるのだが、max_num の処理方法が異なるので比較していきたい。では、比較をしていく前に、使われている関数についての簡単な説明から始めていく。

- calcMax 関数
  - 配列 num に格納された値の最大値を大域変数 max_num に格納する関数である。
  - バイナリモデル検査では、配列 num に任意の値が格納されても clacMax 関数が正しく動作するかを調べている。
- check 関数
  - calcMax 関数が呼ばれた後、max_num には配列 num の最大値が格納されているはずである。check 関数は、本当に配列 num の最大値が max_num に格納されているのかを調べる関数である。

ところで、max_num は適切な処理をしないとどのような振る舞いをするのだろうか。図 3.20 において２９行目の max_num=0 がコメントアウトされているとして考えてみると問題が発生する。 max_num は、まず値 0 で初期化される。 1 回目の calcMax 関数の呼び出し後は、配列 num の最大値が格納される。通常の C プログラムの場合は、ここで終了するので問題はない。しかし、バイナリモデル検査の場合は 3.6.1 項でも述べたように状態が変われば何度も同じ関数が呼び出される。従って、calcMax 関数は再度呼ばれることになる。しかし、2 回目の calcMax 関数呼び出し後に max_num に配列 num の最大値が格納されているとは限らない。1 回目の calcMax 関数呼び出し後の配列 num の最大値が格納されている可能性がある。なぜなら Promela の非決定的な遷移で値を生成し、配列 num に値を格納しているからである。そのため、1 回目の配列 num の最大値の方が 2 回目の配列 num の最大値より大きくなることが十分あり得る。これでは、適切な検査が行えない。このパターンは、図 3.18 に当てはまるの典型的な例である。よって、num_max に対して適切な処理 (検査用コードの構造を図 3.19 に当てはまるようにする処理) を行わなければならない。図 3.20 はその適切な処理を適用した例である。このように前述した一般化した構造と照らし合わせれば、C 大域変数に対しての処理は難しいものではないだろう。

ここで話を少し切り替えて、図 3.20 と図 3.21 の違いについて考察していく。この２つの図の違いは、１２行目と２９行目においてどちらの処理を行っているかだけであるが、その違いは実は大きい。図 3.20 では、max_num に前述した問題があるので clacMax 関数を呼び出す前にはいつも、max_num を 0 で初期化している。これは、元々の C ソースコー

ドにはない記述であるが、正しく検査を行うために追記したものである。図3.21では、そのようにCソースコードに変更は加えていない。その代わりに１２行目でc_trackを使用してmax_numをSPINの内部ステイトベクターに追加している。このようにすることで、SPINにmax_numの値の処理を任せてしまっているのである。

では、この２つの例において記述方法以外に違いがあるのだろうか。記述方法の観点から見れば図3.21の方がスマートである。しかも、全てのC変数をc_trackで指定してしまえば、わざわざ本節のような問題に留意する必要もなくなる。しかし、SPINの内部ステイトベクター用に使用するメモリ量を２つの例で比較すると図3.20の方が効率が良い。図3.20では １２ byte、図3.21では１６ byteのメモリを使用する事が実験で分かっている。どちらの場合も総状態数に違いはなく同一の探索を行っているがメモリ使用量に関して違いが見られた。今回の検査コードはごく小規模であるので、4byte分の違いしかない。しかし、指定するC変数の数や検査の規模により、この値の差が膨れ上がる事は十分に考えられる。従って、 図3.21の記述方法のようにSPINに処理を任せる方法を選択する場合は十分な考察が必要となるだろう。

```
 1 #define MAX 5
 2
 3 c_decl{#include "max.h"};
 4
 5 c_decl{
 6  int num[MAX];
 7  int max_num = 0;
 8 };
 9
10 c_code{#include "max.c"};
11 /*
12 c_track "&max_num" "sizeof(int)";
13 */
14 active proctype test(){
15  byte i=0, r=0;
16 do
17 :: i < MAX ->
18    do
19    :: r < 10 -> r++
20    :: break
21    od;
22    c_code
23     {num[Ptest->i] = Ptest->r;};
24     i++
25    :: else -> break
26 od;
27
28 c_code{
29  max_num = 0;
30  calcMax();
31 };
32 if
33 :: d_step{c_expr{check()} ->
34      assert(false)}
35 :: else
36 fi
37}
```

図 3.20: C 大域変数の処理 (初期化処理を行う)

```
 1 #define MAX 5
 2
 3 c_decl{#include "max.h"};
 4
 5 c_decl{
 6 int num[MAX];
 7 int max_num = 0;
 8 };
 9
10 c_code{#include "max.c"};
11
12 c_track "&max_num" "sizeof(int)";
13
14 active proctype test(){
15   byte i=0, r=0;
16 do
17 :: i < MAX ->
18    do
19    :: r < 10 -> r++
20    :: break
21    od;
22    c_code
23    {num[Ptest->i] = Ptest->r;};
24    i++
25    :: else -> break
26 od;
27
28 c_code{
29 /* max_num = 0; */
30 calcMax();
31 };
32 if
33 :: d_step{c_expr{check()} ->
34      assert(false)}
35 :: else
36 fi
37}
```

図 3.21: C 大域変数の処理 (SPIN に監視させてしまう)

3.7　exit 関数・return 関数の取り扱い

検査したいプログラムには、通常 exit 関数や return 関数が存在する。これらの関数が検査用コード中に存在しても当たり前過ぎて気に留めないかもしれないが、バイナリモデル検査を行うに当たって、検査用コードにこれらの関数が記述されていることは非常に大きな問題である。この問題を説明するにはまず、SPIN の検査方法について簡単に説明する必要がある。

図 3.22 は、SPIN でモデル検査をする場合の流れである。図を見ると分かるように、SPIN を用いて検査する際には、初めに PROMELA 言語で検査用のモデル（example.spin）を記述する。その後、spin -a コマンドで検査用プログラムをコンパイルするのだが、このとき生成される pan ファイルの中身はバイナリではなくて C 言語のソースコードである。その後、pan ファイルをさらに C コンパイラでコンパイルし、検査のために実際に動作するバイナリファイルが生成される。なぜ、このように SPIN そのものがモデル検査を行わないのかというと、メモリ使用量削減のためであったり、検査効率の向上のためであったり、ユーザが pan ファイルを直接ハッキングするためであるのだが詳細は [J.H05] を参照してほしい。以上が SPIN でモデル検査をする場合の流れである。

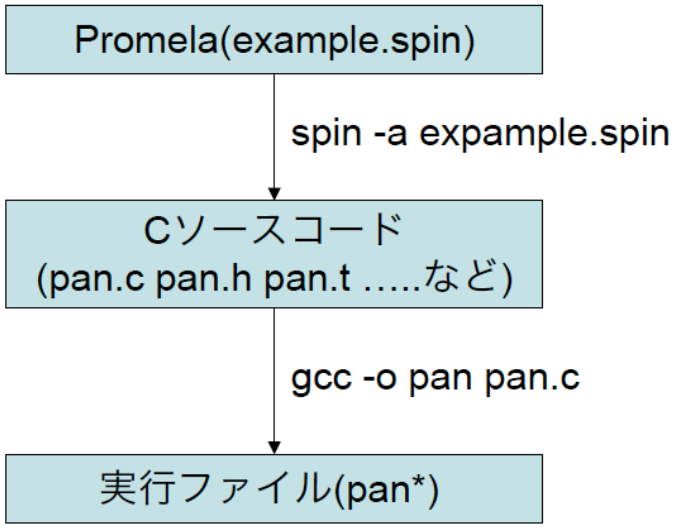

図 3.22: SPIN によるモデル検査の流れ

次にバイナリモデル検査をする場合、埋込 C 機能を利用して PROMELA 上で埋め込んだ C 言語のコードは、pan ファイルに直接埋め込まれる。しかも SPIN 自体は埋込み C に何が記述されているかに全く関与しない。そのため、exit 関数や return 関数を埋め込む

ことが可能となってしまうのである。注意すべき点がここにある。もし、バイナリモデル検査が開始され、検査環境 C プログラムが exit 関数を呼び出すと、検査用に動作している C プログラム pan まで終了してしまうのである。検査プログラムにとっては、予期しない終了であるため反例結果も得ることができず検査が失敗してしまう。return 関数についても同様で c_code 内で return 関数を呼び出すと pan が終了してしまう。しかし、exit 関数と異なり c_code 内で呼び出した関数内で返り値を返す際に使われる分には問題がない。つまり、pan を終了させるような使われ方がされていなければ問題はない。以後、説明で取り上げる return 関数は pan を終了させてしまう return 関数のことであるとする。

そこで、この問題の解決策として exit 関数と return 関数を呼び出すような記述が検査対象のプログラムにあった場合は以下のように対処する。

- 手順
  - c_code 内に直接、exit 関数あるいは return 関数が記述されていた場合：exit 関数あるいは return 関数を Promela の assert(false) に置き換える。どうしても検査をその段階で止めたくない場合 (正常終了なので反例は出したくない場合など) は、適切な Promela 記述と置き換える。
  - c_code 内で呼び出す C 関数が内部で exit 関数を呼んでいる場合：その C 関数の exit 関数をコメントアウトしてしまい、エラー終了の旨を返り値として返すように C 関数を変更する。または C の大域変数を作成しフラグとして利用し、Promela から関数の状態を判定できるようにする。そして、Promela 側でエラー値を処理し、エラーであったら Promela の assert(false) に遷移するように記述する。

以上の対処方法に基づいて、ls プログラムの検査ではどのように実装したかを説明する。

- ls プログラム検査での実装手順
  - c_code 内に直接、exit 関数あるいは return 関数が記述される場合の最も典型的な例は C プログラム終了時の呼び出しである。ls プログラムの検査に際しても、このことは当然あてはまる。よって、ls プログラムにはあるプログラム終了時の exit 関数は Promela にはない。また、正常終了の遷移であるため assert(false) も記述していない。
  - 図 3.24 では、直前で呼び出した関数の終了時の状態を表す値である Ptest → result を c_expr で判定している。もし、Ptest →result がエラー値を示せば、メッセージを出力した後に Promela の assert(false) に遷移するよう C ソースを変更した。このようにすることで、関数の動作による検査プログラムの予期せぬ終了を阻止している。

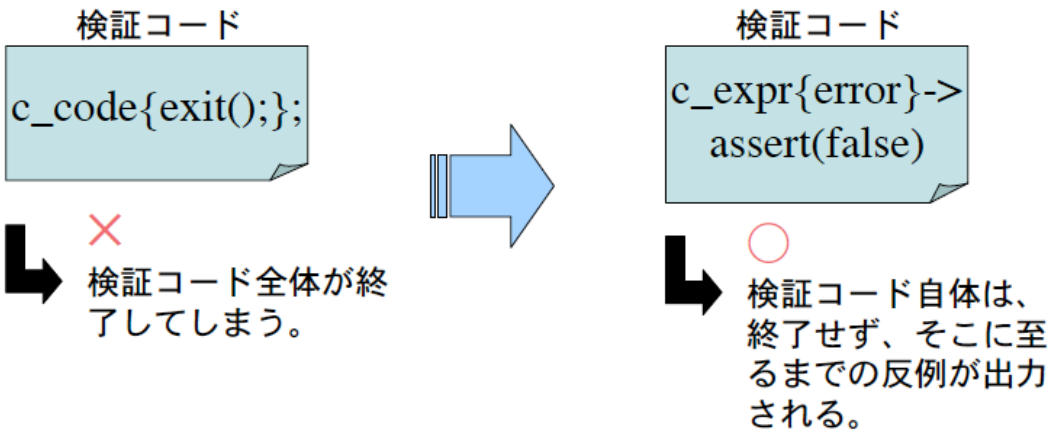

図 3.23: exit 関数置き換えイメージ

```
1 if
2         ::c_expr{Ptest->result == -1} ->
3                 c_code{printf("Error cnt_dir function");};
4                 assert(false);
5         ::c_expr{Ptest->result == 0} ->
6                 c_code{printf("No File");};
7                 assert(false);
8         ::else -> skip;
9 fi;
```

図 3.24: C 変数の値を条件に assert に遷移するよう変更した例

3.8 ポインタ変数の扱いについて

これまで、バイナリモデル検査における様々なC変数の扱い方について述べてきた。この節では、バイナリモデル検査におけるポインタ変数の扱い方について説明する。ポインタ変数をバイナリモデル検査の検査コード上で使用するために、現段階では以下の事項に注意しなければならない。

1. ポインタ変数をSPINの内部ステイトベクターに追加することは可能であるが、SPINによって監視されるのはポインタ変数値であるアドレスであり、参照先のメモリに格納されている値ではない。

    - ポインタ変数ももちろんC変数であるので、埋込Cセグメントであるc_trackやc_stateを使用すれば変数値をSPINで監視することが可能となる。しかし、監視される値はメモリの参照領域ではなくて、ポインタ変数に格納されているアドレス値である。アドレスが示す領域までSPINは、監視しないので注意が必要である。また、アドレス値を監視することが検査においてどのような意味を成すのかについても考えなくてはならない。これは、今後の課題である。

2. SPINが探索木をバックトラックする際に、ポインタ変数が参照しているメモリ領域が存在しない場合があるので注意しなければならない。

    - 図3.25を参照して欲しい。ポインタ変数であるptrがSPINの内部ステイトベクターに組み込まれ探索がなされている様子を表している。注目すべきは、一番右で呼ばれているfree(ptr)である。free関数を呼び出すこと自体は問題ではない。しかし、SPINがバックトラックしてptrを再び参照すると、その領域はすでにfreeされているので問題が発生する。

このようにバイナリモデル検査特有の問題点がポインタ変数に関して存在する。よって、ポインタ変数を不用意にバイナリモデル検査上で使用することは極力控えるべきである。しかし、ポインタ変数が使われていないプログラムしか検査できないのでは不便である。そこで、バイナリモデル検査上でポインタ変数を使用するための解決策を提示する。図4.2がその解決例である。

解決方法として同一のc_code内でポインタの生成、freeを行う。埋込Cセグメントの実行は、SPINに割り込まれる事はなくアトミックに動作する。このことを表したのが図3.27である。図3.25と比較すると分かり易いが、この図では、非決定的な探索パスの選択のあとは決定的に動作するような探索パスとなっている。よって、c_code内でポインタ操作を行えば、ポインタの参照している領域が存在しなくなってしまうという問題も発生しない。c_code内で変化したC変数の状態に関しては、c_codeによって抽象化され、c_code前後の値しか知る事ができなくなってしまうが、安全・確実にポインタ変数を扱える方法である。

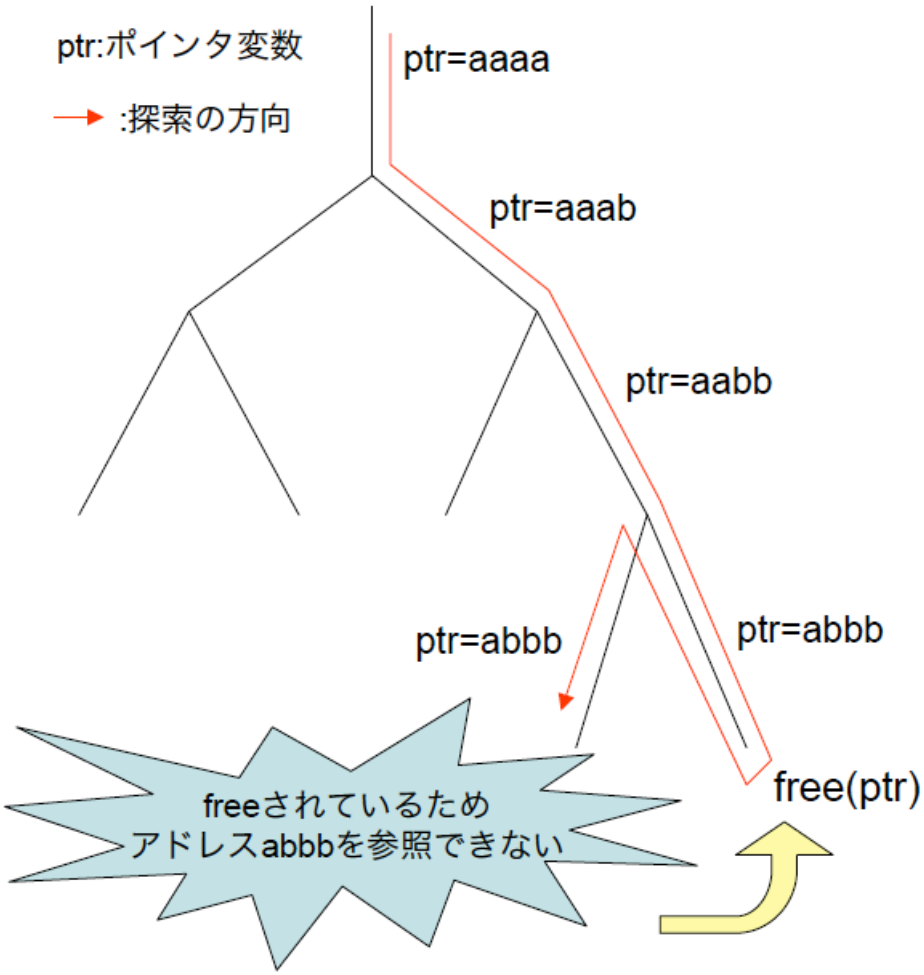

図 3.25: ポインタ変数を監視した際の探索木

今回、解決策を示したが、本来ならば malloc 関数と free 関数の間の対応関係を形式的に記述できるはずである。それが可能であるのならば、malloc 関数と free 関数が別々の c_code 内で呼ばれたとしても、２つの関数間の制約が守られる限り、問題は発生しないはずである。今後、この２つの関数間の対応関係について考察を行いより柔軟にポインタ変数を使用できるようにしていきたいと考えている。また、C 大域変数に関して考察した際の探索木と本節の探索木の間にも何かしらの関係がありそうな気がしている。合わせて考察していきたい。

```
1 c_code{
2 ptr = (int *)malloc(sizeof(int));
3 *ptr++;
4 free(ptr);
5 };
```

図 3.26: ポインタ変数を使用するための解決策

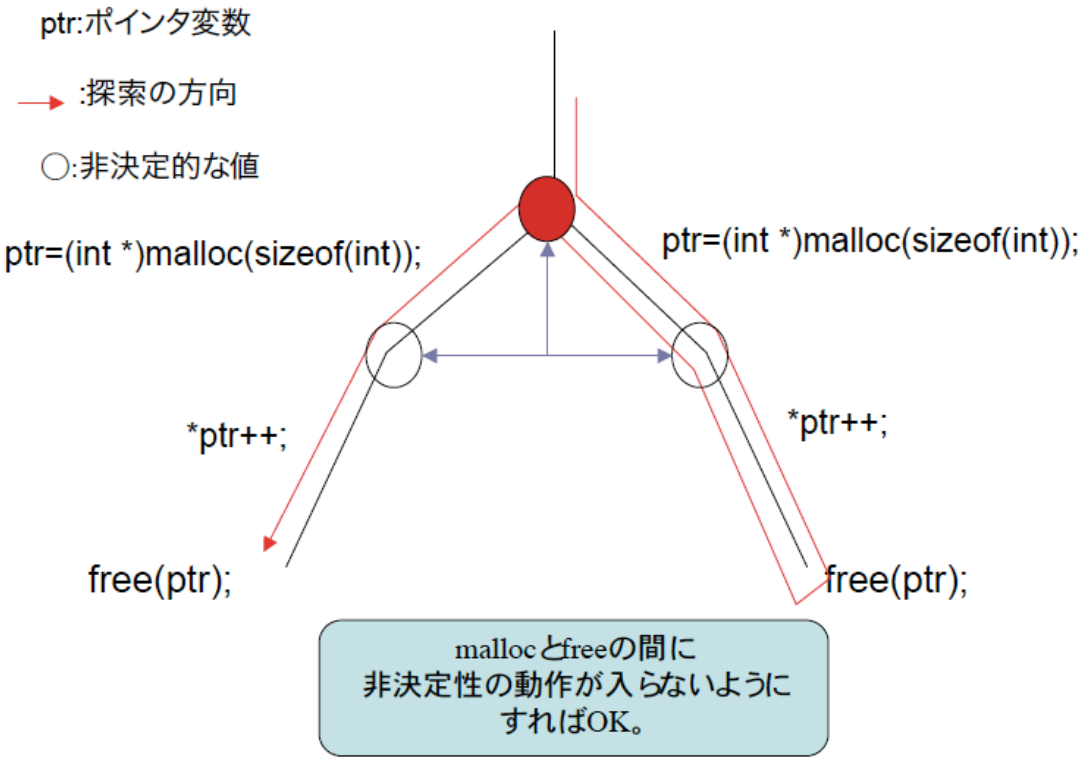

図 3.27: ポインタ変数を使用するための解決策を探索木で表現

第4章　実験結果と考察

本章では、これまで述べてきたバイナリモデル検査を自作 ls プログラムに適用する検査実験について取り上げる。検査内容については、3.3 節を参照してほしい。図 4.1 は、再掲となるが ls プログラムの検査イメージである。この図のようにオプションの組み合わせや読み込むファイル数を入力変数の集合として ls プログラムに与えることによってバイナリモデル検査を行う。

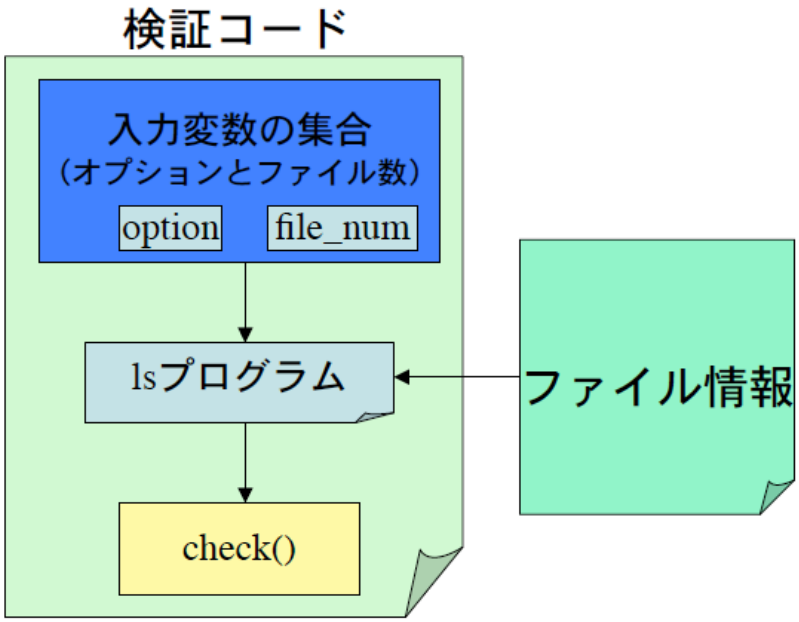

図 4.1: ls プログラム検査イメージ

4.1　ls プログラムの検査結果

本節では、ls プログラムを実際にバイナリモデル検査に適用すると、どのような結果が得られたかについて説明する。また、以後『ls プログラムが正しく動作した』という表現を使うが、これは ls プログラムが正常にファイル情報を出力する遷移に至ることができたということを意味する。

4.1.1　検査環境をそのまま使う場合

検査環境をそのまま使う場合に関しては、任意のオプションの組み合わせを入力変数の集合として ls プログラムに与えるというバイナリモデル検査を行った。その結果、オプションが”-”のみで具体的なオプションが指定されなかった場合、予期しない箇所でプログラムが終了してしまうというエラーが判明した。

4.1.2　検査環境をエミュレートする場合

検査環境をエミュレートする場合に関しては、読み込むファイル数を０～１万まで変化させ、そのファイル情報を入力変数の集合として ls プログラムにあたえるというバイナリモデル検査を行った。その結果、ls プログラムは読み込むファイル数が変化しても正しく動作することが証明できた。

ファイル数の変化に加え、オプションの組み合わせについても検査

検査環境をエミュレートする場合に関しては、読み込むファイル数の違いに加え、オプションの組み合わせについても合わせて検査を行ってみた。その結果、状態爆発が発生してしまった。確かにオプション４つの順列とファイル数１万通りの検査を行えば状態空間が非常に大きくなってしまうことは容易に想像できる。そこで、ファイル数の変化を０～１００に替えて検査を行ってみた。その結果、検査環境が開いている場合で判明したバグを取り除いた ls プログラムならば、正しく動作することが証明できた。

4.2　動的メモリの取り扱い

ls プログラム内では、動的に変数が確保され (calloc 関数呼び出し)、ポインタ変数が使われていた。3.8 節で述べたようにポインタ変数の扱いには十分注意しなければならない。ls プログラムを検査する際においてもこれは同様である。しかし、図 4.2 では 3.8 節で示した解決策を守っていない。同じ c_code 内で calloc と free を呼んでいないのである。なぜ 3.8 節の解決策に従わなかったというと１０行目で呼び出している mysort 関数が Ptest → を参照し、さらにソートにエラーがあった場合反例を出すような Promela 記述をしたかったためである。そのような理由から calloc と free は別の c_code 内にある。しかし、実験を行った結果、問題は全く発生しなかった。なぜなら calloc と free の間に非決定的な遷移が全く無いからである。(if 文中に非決定性遷移があるが行目と１４行目の条件文が通った場合は探索が止まりバックトラックすることがないので非決定的な遷移としてカウントしない。) つまり、calloc と free が組になっていて、その間に他の探索パス上の calloc や free が呼ばれないからである。これを図にすると図 3.27 と全く同じになるのである。つまり、同じ c_code 内に calloc と free が無くても間に非決定的な遷移がなければ良い事が

この実験から判明した。ただし、この条件も十分条件ではあっても必要条件ではない。今後、十分に考察していく必要がある。

```
 1   c_code{Ptest->list = mycalloc(Ptest->file_num);};
 2   c_code{Ptest->seek =
        seek_dir(Ptest->dir,Ptest->list,Ptest->file_num);};
 3
 4   if
 5     ::c_expr{Ptest->seek == -1} ->
 6         c_code{printf("Fail seek_dir");};assert(false)
 7     ::else -> break
 8   fi;
 9
10   c_code{mysort(Ptest->list,Ptest->file_num,
        sizeof(Flist),f_comp,&(Ptest->flag));};
11
12   c_code{Ptest->sort = sortCheck(Ptest->list,
        Ptest->file_num);};
13   if
14     ::d_step{c_expr{ Ptest->sort < 0} ->
        c_code{printf("check = %d",check);}; assert(false)}
15     ::else -> break
16   fi;
17
18   c_code{free(Ptest->list);
```

図 4.2: ls プログラム検査における calloc と free の関係

4.3 ソースコードの変更量

4.3.1 追加したコードについて

ls プログラムの検査実験において追加作成したコードは、オプションの組み合わせを生成するコード（1）、ファイル情報を生成するコード（2）、C 関数列を呼び出すコード（3）、ファイル情報をソートした結果をチェックするコード（4）の 4 種類のコードである。検査環境をそのまま使う場合と閉ざす場合によって（1）あるいは（2）を選択した。ls プログラムの仕様に沿うように、入力変数の集合（1）（2）やチェック関数（4）を作成する作業には手間取ってしまった。なぜならば、PROMELA の非決定的な記述に戸惑ってしまったからである。網羅性を確保できる点は、大きな利点ではあるが PROMELA での記述に慣れていない場合は困難を伴う行程であることが分かった。今後の課題の?? 項でも触れるが、この行程を自動化する仕組みは、バイナリモデル検査を扱いやすくするためにも必要な物なのかもしれない。

4.3.2 元の C プログラムに対しての変更について

C 大域変数の初期化について

ls プログラム検査実験においても注意すべき C 大域変数は存在した。ls プログラムがオプションの判別に使用するフラグが 3.18 項に当てはまる注意すべき大域変数であり、適切な初期化を行わなければならなかった。今回の場合は、定数 0 で全てのフラグを初期化するような関数を新たに作成し、その関数を呼び出す事で問題を解決した。

exit 関数・return 関数について

ls プログラムの場合、内部でシステムコールを呼び出したり、ファイル情報によってはエラーを出力して終了するような関数が多くあったため、PROMELA の assert 関数に置き換える箇所が多くあった。変更の際に大変であったのは、c_code 内で呼び出す関数が内部で exit を呼んでいる場合であった。その関数がもともと返り値を返さないような関数であった場合に、返り値を返すように定義を替えたり、フラグ用の変数を用意するなどの変更を加えなければならず、この作業にはコストが予想以上にかかる事が分かった。この行程においてこのような経験をしたため、どのような C プログラムに対しても対応できる検査方法も大事であるが、検査しやすいように C プログラムを作成する方法論も必要なのではないかと感じた。

4.4 状態数に関して

状態数に関しては、以下の３つの図のような結果になった。興味深かったのは、Spin の内部ステイトベクターで使用したメモリ量に関しては

- 図 4.3(検査環境をエミュレートする場合：ファイル数 (0 から 1 万) を与える) < 図 4.4(検査環境をそのまま使う場合:オプションの組み合わせを与える)

であり、状態に使用したメモリに関しては

- 図 4.3(検査環境をエミュレートする場合：ファイル数 (0 から 1 万) を与える) < 図 4.4(検査環境をそのまま使う場合：オプションの組み合わせを与える)

であったことである。これは、図 4.3 はファイル数の増加が多いが他の入力との組み合わせがなく、図 4.4 は入力変数の値の増加はほぼ無いが他の入力との組み合わせの種類は多いことが原因であると考えられた。また、追加実験として検査をエミュレートする場合にオプションの組み合わせだけでなくてファイル数も変化させるとどうなるかを試してみた。初めは、ファイル数０から１万まで変化させて実験を行ったのだが状態爆発が発生してしまい検査ができなかった。そこで、ファイル数の変化を０から１００までにして実験を再度行った。その結果が図 4.5 である。ファイル数の変化を大幅に減らしはしたが、オプションの組み合わせもあるため、状態ベクトルでのメモリ使用量、状態に使用したメモリ使用量ともに３つの実験の中で最大となった。やはり、入力変数の集合のバリエーションが増えると一気に状態空間が大きくなる。バイナリモデル検査を行う際には、保証したい性質を見極め、変化させる入力変数の選別に時間をかけたい。

```
(Spin Version 4.2.9 -- 8 February 2007)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 48 byte, depth reached 99999, errors: 0
  100000 states, stored
       1 states, matched
  100001 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 343 (resolved)

Stats on memory usage (in Megabytes):
6.000   equivalent memory usage for states (stored*(State-vector + overhead))
5.830   actual memory usage for states (compression: 97.17%)
        State-vector as stored = 46 byte + 12 byte overhead
2.097   memory used for hash table (-w19)
3.200   memory used for DFS stack (-m100000)
3.695   other (proc and chan stacks)
0.101   memory lost to fragmentation
14.923  total actual memory usage
```

図 4.3: 検査環境をエミュレートする場合：ファイル数 (0 から 1 万) を与える

```
(Spin Version 4.2.9 -- 8 February 2007)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 56 byte, depth reached 29, errors: 0
    5993 states, stored
     340 states, matched
    6333 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.929   memory usage (Mbyte)
```

図 4.4: 検査環境をそのまま使う場合：オプションの組み合わせを与える

```
(Spin Version 4.2.9 -- 8 February 2007)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 60 byte, depth reached 2237, errors: 0
  758921 states, stored
     340 states, matched
  759261 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 118760 (resolved)

Stats on memory usage (in Megabytes):
54.642  equivalent memory usage for states (stored*(State-vector + overhead))
51.924  actual memory usage for states (compression: 95.03%)
        State-vector as stored = 56 byte + 12 byte overhead
2.097   memory used for hash table (-w19)
0.320   memory used for DFS stack (-m10000)
0.123   memory lost to fragmentation
54.334  total actual memory usage
```

図 4.5: 検査環境をエミュレートする場合：オプションの組み合わせとファイル数 (0 から 100) を与える

4.5 関連研究 FeaVer について

本節では、関連研究である FeaVer について簡単に説明しておきたい。FeaVer は SPIN 同様、L.T & Bell 研究所で開発された検査ツールである。FeaVer は、内部で SPIN の埋込 C 機能を利用して C プログラムを検査している。この説明だけでは、バイナリモデル検査と同じ事をしているように感じてしまう。しかし、FeaVer が保証しようとしている性質はバイナリモデル検査とは大分違う性質である。このことを説明するには、FeaVer の検査の流れを知った方がよい。そこで、まず FeaVer の構成について説明していく。

4.5.1 FeaVer の構成

FeaVer は、図 4.6 のように構成されている。まず、ユーザは Test harness と呼ばれるファイルに C プログラムと PROMELA の対応表を作らなければならない。対応表には、どの関数がどのプロセスに対応するか、どの C 変数を SPIN の内部ステイトベクターに組み込むか、などについて記述する。次に、記述された Test harness と C プログラムを Modex と呼ばれる自動モデル抽出器に読み込ませる。すると Modex は、Test harness をもとに埋込 C 機能を使っている PROMELA を出力する。あとは、SPIN がこの PROMELA を実行するだけである。

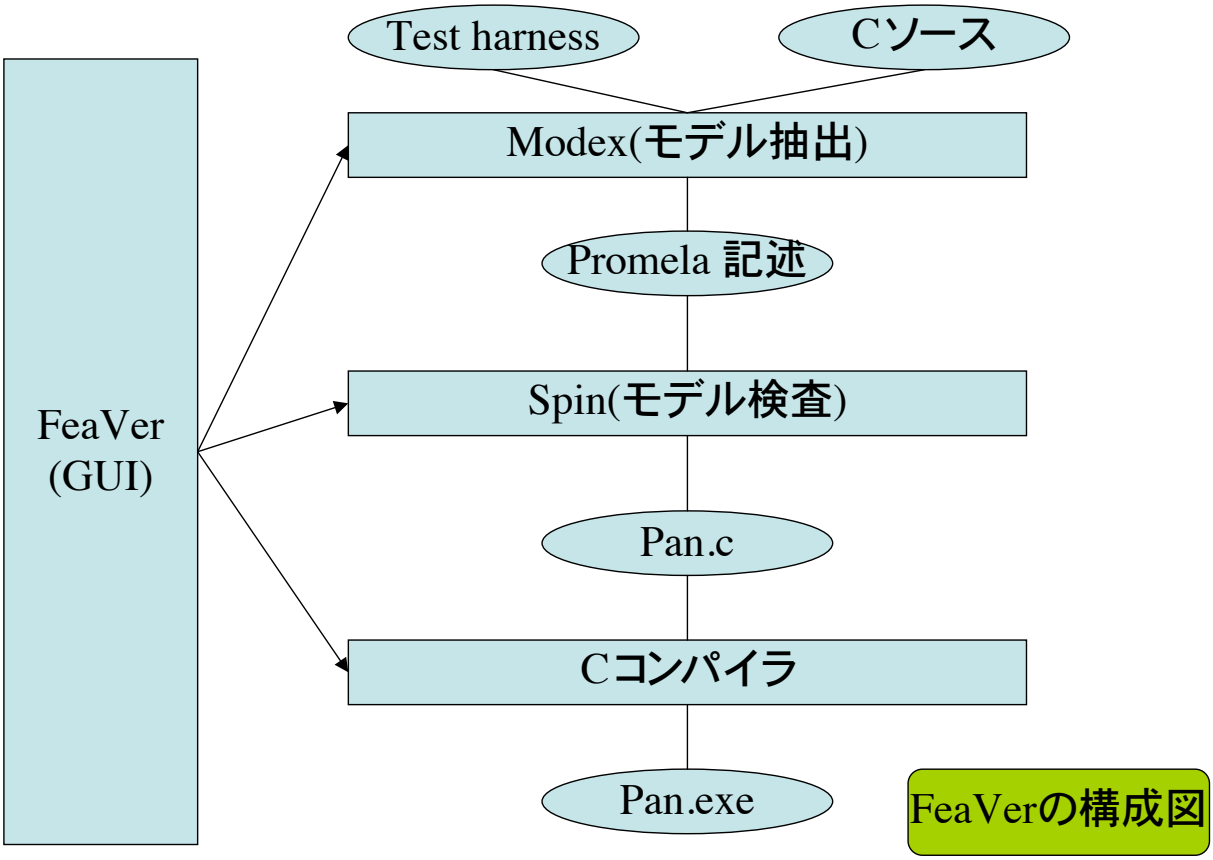

図 4.6: FeaVer の構成

4.5.2 FeaVer とバイナリモデル検査の違い

FeaVer は並行性をもつ C プログラムの検査をターゲットにしている。なぜなら Modex は、C の関数を PROMELA のプロセスに対応付けし、デッドロックや競合状態を追跡できるようなモデルを生成することに秀でているからである。それに、対してバイナリモデル検査では、シーケンシャルに動作するプログラムを保証しようとしている。入力変数の集合を与えることで呼び出される C 関数列がどのように動作するかを調べるからである。また、動作環境を積極的にそのまま利用する点もバイナリモデル検査の特徴であり FeaVer とは異なっている。

4.5.3 FeaVer の利点と不利な点

Test harness さえ適切に記述できれば、あとは C プログラムを全自動で検査できることがやはり FeaVer の大きな利点である。配列の境界条件チェックや NULL ポインタチェックを行う c_code をデフォルトで追加してくれることも良い機能である。しかも、GUI で操作できるので SPIN のオプションの設定などは非常に分かり易い。しかし、FeaVer を使っての初めのである Test harness 作成についての理解が難しい。この行程を適切に行うためには、SPIN の機能の正確な理解が求められるであろう。つまり、FeaVer は SPIN に関して知識豊かな人のためのツールであり、モデル検査に関して初心者が人に使いこなす事は難しいと考えられる。これが不利な点である。さらに、FeaVer のマニュアルおいて動的に確保したメモリ領域の扱い方については全く触れられていない。今後、バイナリモデル検査において、動的に確保したメモリ領域の監視や局所変数の監視が可能となれば FeaVer に対する大きなアドバンテージになると考えている。

第5章　おわりに

5.1　まとめ

本研究では、バイナリモデル検査という独自手法を提案し、様々な問題の解決と整理を行った。まず、バイナリモデル検査手法の概要を整理し、どのような検査手法であるのかを明確化させた。そして、本検査手法を行うことによって得られる利点について考察した。バイナリモデル検査を行うにあたり、検査対象 C プログラムを検査可能な形に変更するために内容変更が必要な場合があるので、その変更必要箇所と変更手順を手法として提案した。次に、このように整理・一般化した検査手法を検査実験として sort プログラムや ls プログラムなど実際のプログラムに適用し、その効果について考察した。結果として、ある程度の規模の C プログラムであっても、多少の関数の仕様変更は迫られるものの、プログラムフローに関しては、ほとんど変更せずに Spin から直接呼び出して検査可能であることが分かった。以上が検査法確立に関しての成果である。またそれに関連して、技術面に関しての成果もいくつかあった。監視する C 変数指定方法とその扱い方に対する考察や、効率的にメモリ空間を監視するための工夫、C プログラム中で使用される動的メモリ空間の取り扱い方に対する問題提起と解決策の提案などが代表的な技術的成果である。

これらの研究成果からバイナリモデル検査を利用すれば、検査コストが少なく、実環境により近い状態での C プログラム検査が可能であることが示せた。

5.2　今後の課題

これまでに、バイナリモデル検査を行うにあたっての問題に対する解決策をいくつか提示してきた。しかし、本手法をより確立したものにするために、解決すべき課題がまだある。筆者は、同学博士後期課程に進学する予定であるので、以下に示した課題について積極的に今後、取り組んでいきたいと考えている。

5.2.1　バイナリモデル検査の自動化

図 5.1 のようにバイナリモデル検査を自動化したいと考えている。

- 入力変数の集合の形式的な定義と自動生成
  sort プログラムや ls プログラム、その他例示したプログラムをバイナリモデル検査す

る際、検査概要に従って入力変数の集合を与えた。しかし、現段階でこの工程では、個々の問題に対して人が考え、入力変数の集合にあたるモデルを手作業で PROMELA で記述している。しかし、検査方法をより形式化するためや検査の効率を上げるためにも、この工程を自動化すべきであると考えている。少なくとも入力変数の集合を数学的に定義し、論理式で入力変数の集合を表現できるようにすることはしなければならない。理想的には、形式的に記述されたプログラム仕様書と入力変数の集合を表した論理式がさえあれば、自動で入力変数の集合のモデルを記述した PROMELA のコードが生成できれば良いと考えている。

- 出力変数の集合に対するチェック関数の仕様記述に基づく自動生成
  チェック関数に関しても自動生成を行えれば良いと考えている。検査概要では、入力変数の集合が C 関数呼び出し列によって処理され、出力変数の集合が得られ、その集合をチェック関数で調べる。このチェック関数を上記と同様に論理式とプログラムの仕様書から自動生成できれば、C ソースコード、プログラム仕様書、論理式さえあればバイナリモデル検査が自動で行えるようになり非常に魅力のある機能になると思われる。

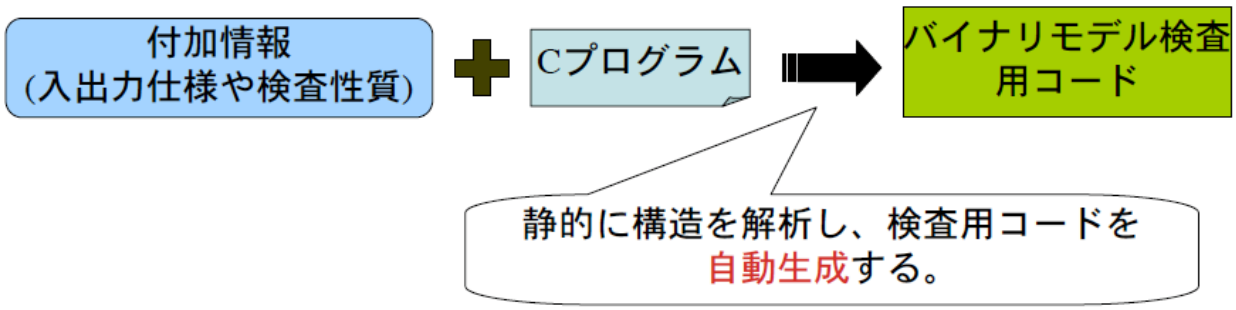

図 5.1: バイナリモデル検査を自動化する

5.2.2 c_code の使い方について

本論文では、c_code の使い方については特に言及してこなかった。しかし、c_code は埋込 C 機能の最も基本的なフラグメントであるので適切に使用することが出来れば検査効率にも非常に大きな効果を与えると考えられる。よって、その適用方法についていずれ考察しなければならない。その考察すべき課題のひとつが、どの程度の粒度で C ソースコードを囲むのかについてである。現段階では特に問題がない限り、c_code 内の C ソースコードの行数を少なくしている。なぜなら、そうすると SPIN 側から詳細な C 変数の情報を得る事ができるからである。しかし、その分状態数も増え効率的ではない。検査に差

し障りのないぎりぎりの範囲を c_code を囲むことができれば、無駄な状態数を減らしつつ効果的な検査ができるようになる。実は、これと同様なことが通常の SPIN を使ったモデル検査においてもよく行われている。d_step や atomic である範囲をアトミック実行することで状態数を減らすという、一種のプログラミングテクニックである。c_code もアトミック実行であるので同様の効果をもたらす。ただし、PROMELA に加え C の振る舞いの意味も考えなければならないので十分な議論が必要である。

5.2.3 C 大域変数の初期化に関する議論

3.5 節において、C 大域変数の取り扱いに議論し、初期化が必要な C 大域変数の発見方法及び対処方法について述べた。しかし、実は 3.5 で示した C 大域変数の発見方法は十分条件ではあるが必要条件ではない。つまり、現在の発見方法を C 大域変数に適用すると、注意すべき全ての C 大域変数はフィルタにかかるのだが、注意しなくても良い C 大域変数までフィルタにかかる可能性があるのである。過剰な C 大域変数の監視は、メモリの使用量を無駄に増加させてしまう要因である。少しでも無駄な指定をなくし、検査コストを減らすためにも議論を行い、注意すべき C 大域変数発見のための必要十分条件を見つけたいと考えている。

5.2.4 動的に確保される変数、局所変数に対する SPIN からの監視手法

4.2 節において、動的に確保される変数の扱いについて述べたが、現在のバイナリモデル検査では、検査が開始された後に確保された変数（動的変数、局所変数）については SPIN の内部ステイトベクターに組み込むことができない。つまり、組み込めるようにするモデル検査器を自作するか、それを代替するような解決策を提案する必要がある。ここでは、後者の場合の考察中解決策について説明したいと思う。図 5.2 は、C バイナリが実行され、SPIN が割り込み可能になった時点で C 変数の状態をチェックしようとする図である。しかし、図のように c_code 内で後から生成された変数を SPIN は監視することはできない。そこで、SPIN からではなく、C プログラム側から変数値を SPIN 側のあらかじめ用意された内部ステイトベクターへ定期的にコピーする仕組みが作れないだろうかと考えている。そのようにすれば、SPIN が割り込んだときに今まで見る事のできなかった c_code 内の値を知る事ができ、過去の値にバックトラックすることも可能なのでないかと考えている。ただ、この方法では C 変数の全ての情報をコピーしてしまうのでメモリ使用量の観点から言えば効率的ではない。そこで、C 変数の値が何らかの意味 (値が負になるなど) をもつときだけコピーすることが可能になれば、変数情報を抽象化することにもなり、メモリを有効的に使用できるようになるのではないかと考えている (図 5.3)。

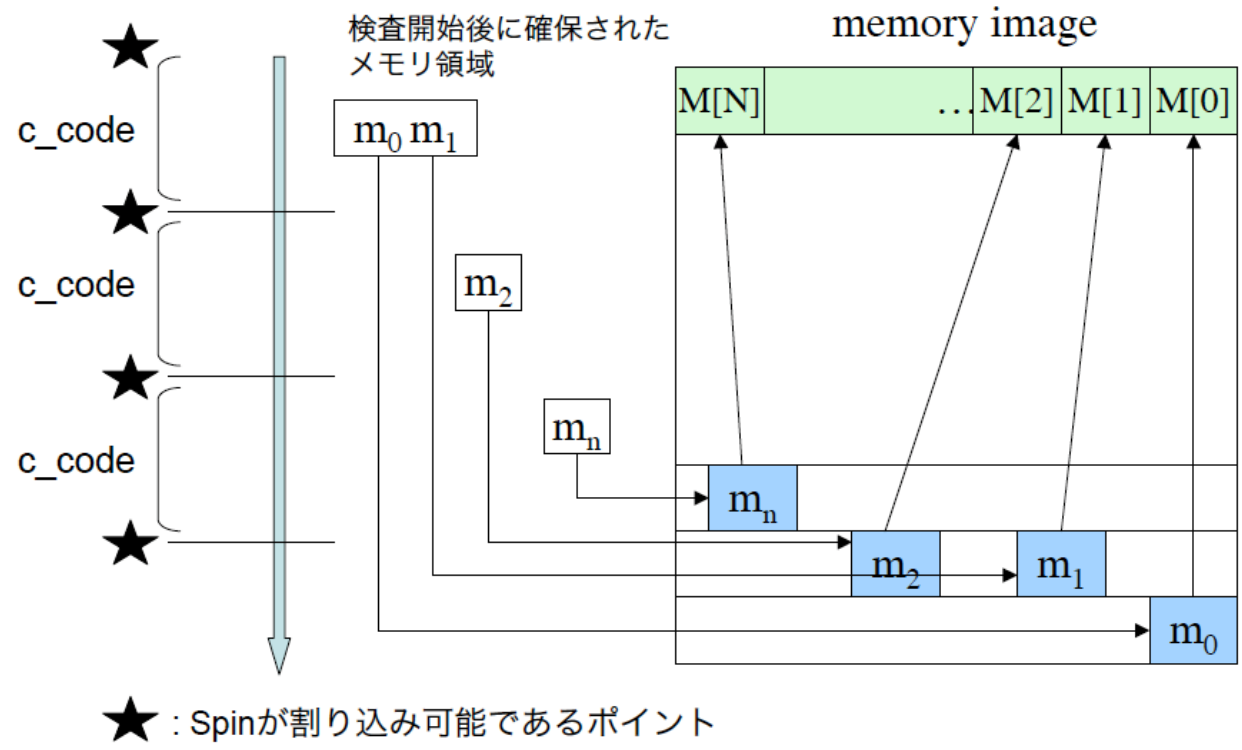

図 5.2: 後から確保されたメモリ領域の情報を SPIN 側へコピーする

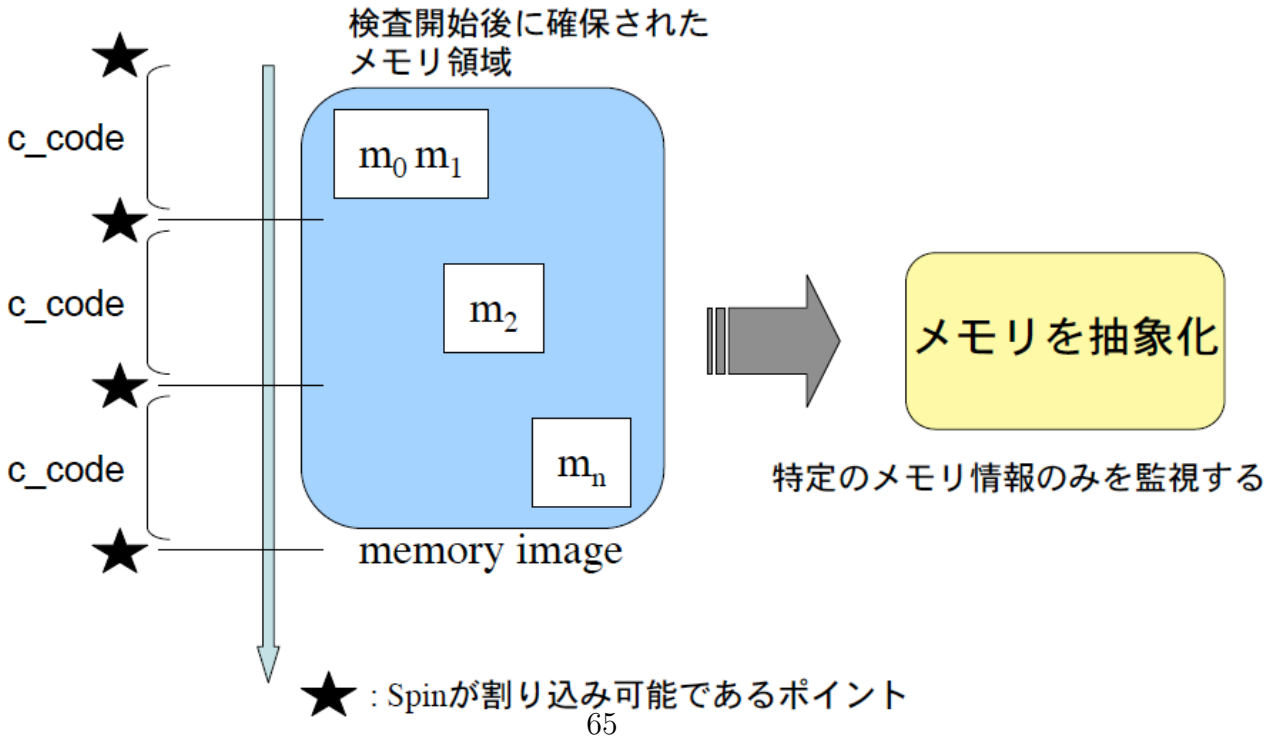

図 5.3: 後から確保されたメモリ領域の情報を抽象化して SPIN から監視する

謝辞

北陸先端科学大学院大学　安心電子研究センター　青木利晃特任准教授には、本研究を進めるにあたり直接の御指導を頂き、ここに感謝の意を表明したいと思います。

また、有益な助言をして頂いた、同学　片山卓也教授、岸知二特任教授に厚く御礼を申し上げます。

その他、貴重な御意見、討論を頂き、共に研究生活を過ごす事ができた青木研究室、岸研究室、片山研究室、Defago 研究室の皆様に深く感謝いたします。

最後に、研究生活を様々な面で支えてくださった家族に感謝します。

参考文献

[HS] Gerard J. Holzmann and Margaret H. Smith. *FeaVer1.0 User Guide*.

[J.H05] Gerard J.Holzmann. *THE SPIN MODEL CHECKER*. Addison-Wesley, 2 edition, 2005.

[田辺 04] 田辺良則, 高井利憲, 高橋孝一. 抽象化を用いた検証ツール, 2004.

付 録A　lsプログラムを構成している関数詳細

＜myls.cについて＞
myls.cは、ファイル表示に必要になる情報を指定されたパスからとりあえず全て獲得し、構造体配列Flistに保持しておく。その後、オプションに応じて対応する関数を呼び出し構造体配列の順番を変えたり、表示の仕方を変えたりする。

構造体：Flist
説明：　stat関数から得たファイル情報を一時的に保持しておく構造体

構造体：Flags
説明：　どのオプションが指定されたか記憶するためのフラグ

関数名：main
引数：　int型、char型へのポインタ
返り値：int型
動作：　指定したディレクトリのファイル表示
説明：　main関数は、大きく分けて４つの働きをする。まず、指定されオプションとパスをif文、switch-case文などで判別する。パス指定されなかった場合はカレントディレクトリを走査する。次に、指定されたディレクトリのファイル数を調べ０ならエラー処理（そんなディレクトリはないことを意味している)、１つでもあるならディレクトリの内容を調べ始める。そして、オプションの状況に応じてソート(何も指定されなかったときはファイル名でソートする)をする。最後に、ファイル情報を表示し終了する。

関数名：cnt_dir
引数：　char型へのポインタ
返り値：int型
動作：　指定されたディレクトリないのファイル数をカウント
説明：　引数に指定された文字列（パス名）からディレクトリオープンし、openすることができたらディレクトリ内のファイル数（ディレクトリ含む）を数える。ただし、"."と".."は数えない。数え終わるとディレクトリをcloseしファイル数を返す。

関数名：seek_dir

引数： char 型、Flist 型へのポインタ、int 型
返り値：int 型
動作： ファイル情報の獲得
説明： 指定されたディレクトリ内のファイル情報を stat 関数を用いて獲得し、調べたファイルがディレクトリであった場合は文字列に"/"を足す。そして、必要な情報を list_cpy 関数を使用して構造体 Flist にコピーする。この一連の作業が成功したら 0 を失敗したら-1 を返す。

関数名：list_cpy
引数： Flist 型へのポインタ、構造体 stat
返り値：なし
動作： stat の内容を Flist へコピーする
説明： この関数は、seek_dir 関数内で呼び出される。構造体 stat が保持するファイル情報をソートなどで利用するために保持しておくと都合が良いので、この関数で必要な情報を Flist にコピーする。

関数名：mysort
引数： Flist 型へのポインタ、size_t 型*2、関数 comp へのポインタ、構造体 Flags へのポインタ
返り値：なし
動作: 内部的に qsort を呼び出して、Flist 構造体配列をソートする
説明： この関数は、Flags に応じて比較する条件を変えて qsort 関数を呼び出す。qsort に与える比較のための関数は全部で 4 つ。
ファイル名比較と時間比較、そして、それぞれに対して逆比較の関数がある。Flist に対して直接ソートを行うので返り値はない。

関数名：t_comp
引数： const void 型* 2
返り値：int 型
動作： 引数を比較する
説明： この関数は、qsort 関数に使われる比較関数。構造体 Flist のメンバ time_str を比較する。a が b より 1) 小さい 2) 等しい 3) 大きい場合に、ゼロよりも 1) 小さい 2) 等しい 3) 大きい数を返す。

関数名：t_comp_r
引数： const void 型* 2
返り値：int 型
動作： 引数を比較する
説明： この関数は、qsort 関数に使われる比較関数。構造体 Flist のメンバ time_str を比較する。a が b より 1) 小さい 2) 等しい 3) 大きい場合に、ゼロよりも 1) 大きい 2) 等しい 3) 小さい数を返す。

関数名：f_comp
引数： const void 型* 2
返り値：int 型

動作： 引数を比較する
説明： この関数は、qsort 関数に使われる比較関数。構造体 Flist の
メンバ f_name を比較する。a が b より 1) 小さい 2) 等しい 3) 大
きい場合に、ゼロよりも 1) 小さい 2) 等しい 3) 大きい数を返す。

関数名：f_comp_r
引数： const void 型*２
返り値：int 型
動作： 引数を比較する
説明： この関数は、qsort 関数に使われる比較関数。構造体 Flist の
メンバ f_name を比較します。a が b より 1) 小さい 2) 等しい 3) 大
きい場合に、ゼロよりも 1) 大きい 2) 等しい 3) 小さい数を返す。

関数名：get_permission_srt
引数： Flist 型へのポインタ
返り値：なし
動作： アクセス許可を表す数値を文字に変換する
説明： この関数は、stat 構造体の st_mode の値を保持した list->mode の
値を調べ、drwxrwxrwx のような形式に変換する。内部的には、
if 文でフラグとの論理積をとり判定、文字列を構成している。
最後に、Flist 構造体に書き込んでおく。

関数名：print_files
引数： Flist 型へのポインタ、int 型、Flags 型へのポインタ
返り値：なし
動作: ファイル情報を標準出力に出力する。
説明: この関数は、オプションの情報がある Flags のメンバを調べ、そ
れに応じた出力をする。ほとんどの情報は、Flist にあるので
そのまま出力している。けれど、ユーザ名とグループ名は、ユー
ザ ID とグループ ID をもとに、それぞれ passwd 構造体、group 構
造体を取得し、そのメンバの情報を表示しています。

関数名：usage
引数： なし
返り値：なし
動作： 使い方の表示
説明： この関数は、間違ったオプションが指定されたとき main 関数内で
呼び出される。myls コマンドの使い方、オプションの種類を標
準出力に出力してプログラムを終了する。

＜ time.c について＞
time.c は、stat 構造体から取得した時間情報を理解しやすい形式に変換し
て文字列として返す関数が 1 つ定義してるだけ。

関数名：get_time_str
引数： time_t 型
返り値：char 型へのポインタ
動作： time_t 型を引数にして解釈、文字列として返す関数
説明： まず、localtime 関数を使用して UNIX 時間を"年月日時分秒"に変更。strftime 関数によって指定したフォーマットの文字列を calloc で取得した領域に格納し、返す。

＜ err.c について＞
err.c はエラー処理をする関数群がメイン。関数の種類は Fatal Error かどうか、System Call に依存したエラーかどうか、強制終了するのか return するのか、といった具合いでにエラー要因に応じて処理を使い分けられるように作成したもの。

＜ myhdr.h について＞
myhdr.h は include 文、マクロ、プロトタイプ宣言などを纏めたもの。
便利な関数が多いので自分用のヘッダファイルとして使い回しているもの。

＜実行例＞
--オプション、パスを指定しなかった場合

```
#[s0610060@lss6-ws03] 34 % ./myls
  Makefile
  compare.c
  err.c
  myhdr.h
  myls
  myls.c
  report.txt
  report.txt~
  time.c
  tmp/
```

--オプションのみ指定した場合（全ての情報を表示して逆順に時間ソートするオプション）

```
#[s0610060@lss6-ws03] 60 % ./myls -iltr
inode        permission      links   u_ID    g_ID    file size       last modefied   file

65187829     -rw-r--r--      1       s0610060        is      9904    2006 11/02 19:43:53
65187841     -rw-r--r--      1       s0610060        is      892     2006 11/02 19:43:53
65187841     -rw-r--r--      1       s0610060        is      892     2006 11/02 19:43:53
65187835     -rw-r--r--      1       s0610060        is      1984    2006 11/02 19:43:53
24053030     -rwxr-xr-x      1       s0610060        is      16008   2006 11/02 19:43:53
65187840     -rw-r--r--      1       s0610060        is      9314    2006 11/02 19:32:27
65187832     -rw-r--r--      1       s0610060        is      3157    2006 11/02 10:43:58
24053037     -rwx------      1       s0610060        is      4105    2006 10/31 17:10:51
```

```
  24053031     -rw-r--r--      1       s0610060        is      197     2006 10/31 16:43:34
  24053033     -rw-r--r--      1       s0610060        is      414     2006 10/31 16:37:35
  24053034     -rw-r--r--      1       s0610060        is      1750    2006 10/31 15:52:55
  24053024     drwxr-xr-x      2       s0610060        is      4096    2006 10/31 11:38:52
  65187837     -rw-r--r--      1       s0610060        is      405     2006 10/31 11:37:59
--パスのみ指定した場合（カレントディレクトリ指定）
  #[0610060@lss6-ws03] 62 % ./myls .
  Makefile
  compare.c
  err.c
  myhdr.h
  myls
  myls.c
  report.txt
  time.c
  tmp/
--パスとオプションを指定した場合（inode 以外の情報を表示）
  #[s0610060@lss6-ws03] 73 % ./myls -l ~/softwear_toku/report2/
  permission   links   u_ID    g_ID    file size       last modefied   file

  -rw-r--r--   1       s0610060        is      197     2006 10/31 16:43:34     Makefile
  -rw-r--r--   1       s0610060        is      414     2006 10/31 16:37:35     compare.c
  -rw-r--r--   1       s0610060        is      1750    2006 10/31 15:52:55     err.c
  -rw-r--r--   1       s0610060        is      1984    2006 11/02 19:43:53     err.o
  -rwx------   1       s0610060        is      4105    2006 10/31 17:10:51     myhdr.h
  -rwxr-xr-x   1       s0610060        is      16008   2006 11/02 19:43:53     myls
  -rw-r--r--   1       s0610060        is      9314    2006 11/02 19:32:27     myls.c
  -rw-r--r--   1       s0610060        is      9904    2006 11/02 19:43:53     myls.o
  -rw-r--r--   1       s0610060        is      4491    2006 11/02 19:58:06     report.txt
  -rw-r--r--   1       s0610060        is      3157    2006 11/02 10:43:58     report.txt~
  -rw-r--r--   1       s0610060        is      405     2006 10/31 11:37:59     time.c
  -rw-r--r--   1       s0610060        is      892     2006 11/02 19:43:53     time.o
  drwxr-xr-x   2       s0610060        is      4096    2006 10/31 11:38:52     tmp/
--存在しないディレクトリを指定した場合
  #[s0610060@lss6-ws03] 74 % ./myls -l ~/hoge
  error open dir
  Error cnt_dir function
--存在しないオプションを指定した場合
  #[s0610060@lss6-ws03] 75 % ./myls -k
  Unknown option -?
  Usage: ./myls [-options] /path/
  options:
       -i      print number of inode
       -l      print file's information
       -t      time sort     (default:file name)
       -r      reverse sort
```