9. モデルの抽象化と状態爆発の対策法

本章では、状態爆発の全体像を示し、その原因と解決策について具体的な方法を例示することにより、状態爆発を回避する方法を示す。本章の概要は以下のとおりである。

対象読者	開発技術者、(開発プロジェクト管理者)
目的	モデル検査における状態爆発の技術的問題点を示し、その解決策として、モデルの抽象化等の対策法を示す。特に、既存の書籍ではあまり書かれていない、Promela のコードレベルでの具体的な対策について示す。
想定知識	SPIN の入門書などによりモデル検査の入門知識を持つソフトウェア技術者。
得られる知見等	● 状態爆発の問題点と原因の具体例 ● 抽象化の概念と留意点 ● 状態爆発の対策の全体像 ● Promela コードレベルを中心に具体的な対策法

9.1. 状態爆発の問題点

モデル検査における技術的な制約として、「状態爆発」と呼ばれる状況で、自動検査が長時間終了しない問題が発生することがある。大規模なソフトウェアの詳細設計をそのままの詳細度でモデリングし、モデル検査を行うと、状態爆発を発生する可能性が高い。

SPIN の場合、モデルの状態は、大まかに変数の値、並行プロセスのステートメントの実行位置の組合せ、通信チャネルのバッファ内の値などの組合せによって決まる。Promela のコード量が少なくても、これらの組合せが莫大になればすぐに状態爆発を発生するため、必ずしもコード量だけからでは判断が出来ない。

例えば、以下のような単純な2つの並行プロセスを考えた場合、SPIN においては、それぞれのプロセスの状態遷移は、非同期に並行実行されるとみなされる。

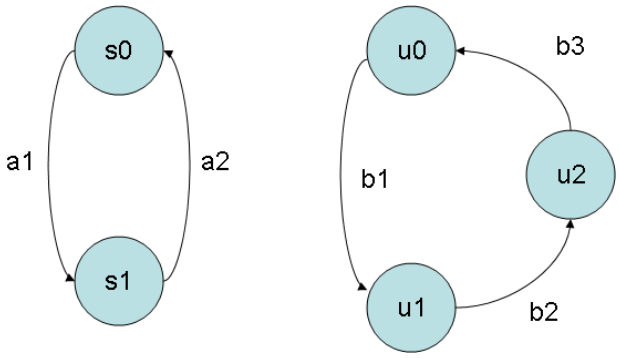

図 9-1:2つの並行プロセスの状態遷移図(例)

プロセスが非同期並行実行されると、各プロセスの実行ステートメントは、任意の組合せで実行(インターリービング)されるため、全体のプロセスはその組合せにより状態が増大する。上記の2つの並行プロセスから構成される全体の状態遷移システムは、下図のようになる。

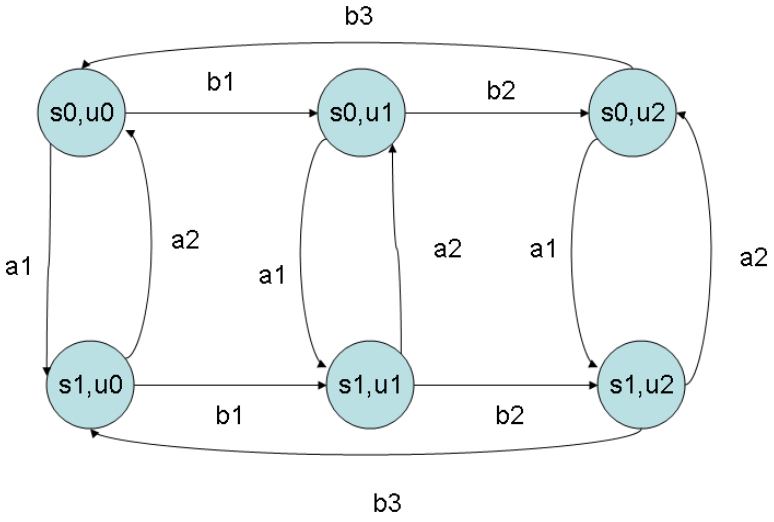

図 9-2:非同期並行実行による状態遷移図

一方、変数の数が増えれば、変数が取りうる値の数を変数分掛け合わせた変数の数の指数のオーダーで状態数が増大する。例えば、整数(2^32)の変数が n 個使われると、大まかに(2^32)^n のオーダーで増加する。

以上のような指数爆発が発生するようなモデルは容易に状態爆発が発生し、モデル検査が終了しない。このような状況に対処するために、モデルを抽象化することで、状態爆発を回避する必要がある。ただし、モデルの抽象化は、状態爆発を回避することだけが目的ではない。それについて以下の節でまとめる。

9.2. 抽象化に関する留意点

モデルの抽象化は、検証したい性質などのある観点に注目して、それに無関係な情報を捨象

することである。例えば、簡単な例で言えば、システムのうち検証したいコンポーネントと無関係なコンポーネントを除外することは一つの抽象化である。また、もう少し難しい例を挙げると、ソフトウェアのうち、何を実現するか(What)を表す「要求」に注目して、機能をどのように実現するか(How)を表す実現方法を、取り除くことも抽象化である。また、ある制御アルゴリズムにおいて、注目したデータに影響を与えるステートメントや制御構造のみを抽出するスライシングと呼ばれるものも、抽象化である。

抽象化は、状態爆発を回避することなどを含め、以下のような目的に分けられる。

目的	内容
状態爆発の回避 (技術上の制約回避)	検証性質に関係の無いモデルの要素を捨象することで、状態数を削減する。
検証の効率化 (作業コストの低減)	検証したい内容に関係の無い部分を捨象して、作業コストを低減する。
ユーザ要求レベルの記述	実装に依存しないユーザの要求の視点に近いレベルで性質を記述し、要求とのギャップによる不具合の混入を防止する。

プログラミングでは、あまり抽象化の概念は出てこないため、習得の難しい概念であるが、状態爆発という技術的制約を回避するだけでなく、問題の本質を捉え、ユーザ要求など上位の概念を記述することで、本来の目的に近い観点から検証を行うことによる効果が期待できる。

状態遷移システム(モデル)の抽象化の具体例を示すために、非常にシンプルな以下の例を考える。

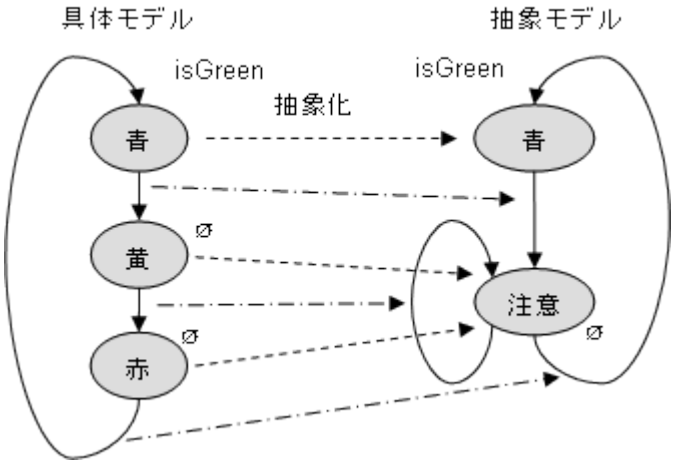

図 9-3:交通信号モデルの抽象化(例)[150]

状態遷移システムにおける抽象化は、状態と遷移の縮退として捉えられる。図 9-3 の左の具体モデルは、交通信号の「青」「黄」「赤」の状態遷移を表している。それを抽象化した右側の抽象モデルは、「黄」「赤」の2つの状態を「注意」という1つの状態に抽象化し、それに対応して状態遷移を修正したものである。

モデル検査を行う場合、モデルの抽象化において、検証したい性質が保存されるかどうかが重要になる。抽象化したモデルを検証する際には、通常、以下のようなことが期待される:

- 抽象化の健全性
  モデル検査により抽象モデルで成立つことが示された性質は、具体モデルでも成立つ。
- 反例の保存性
  モデル検査により抽象モデルで反例が検出された場合、具体モデルでも反例が存在する。

上記のモデル抽象化について以下のような性質について考える。

「いつでも信号は、いずれ「青」以外に切り替わる。」 (9-1)

図 9-3 において、状態「青」でのみ成立つ命題を isGreen とすると、この論理式は、以下のように記述される。

[] <> I isGreen

この性質は、抽象モデルで成立ち、具体モデルでも成立つことが、簡単な思考によりすぐ分か

[150] 田辺、高井、高橋: 抽象化を用いた検証ツール, コンピュータソフトウェア, Vol. 22, No. 1, pp. 2-44, 2005.

る。
一方、以下のような性質は、具体モデルでは、状態「注意」への自己ループにより無限遷移し、「青」に遷移しない場合もあるため、具体モデルで成立ちませんが、抽象モデルでは成立つことが分かる。

「いつでも信号は、いずれは「青」に切り替わる。」(9-2)

このようなことから、例のモデルの抽象化では、少なくとも、「抽象モデルに反例がある場合、具体モデルでも反例がある。」という性質は成り立たない。

一般に、抽象化の健全性が成立つためには、以下のような3つの条件が成立てば良いことが証明されている[151]。

図 9-4:抽象化の健全性の条件(言葉による説明)

抽象化の健全性の条件

(1) 具体モデルの初期状態を抽象モデルに写像した状態は、抽象モデルの初期状態集合に含まれる。

(2) 具体モデルで遷移関係のある2つの状態は、抽象モデルに写像した先でも、遷移関係にある。

(3) 具体モデルの状態で成立つ命題全体は、抽象モデルに写像した状態で成立つ命題全体と一致する。

この条件が成立てば、特定の時相論理式のクラス(ACTL*)の任意の論理式について、抽象モデルで成立つことは、具体モデルでも成立つことが示されている[152],[153]。条件の正確な定義は、それらの文献を参考にされたい。

健全性の条件が成立っていても、反例の保存性は一般には成り立たない。その例が、数式(10-2)の例が示している。抽象モデルで反例が見つかった場合、通常、以下のような対処を行う。

(1) 抽象モデルの反例の実行系列から、具体モデルの実行系列で反例になるか分析する。
(2) 抽象モデルの反例から、具体モデルの実行系列が反例ではない時(偽反例)、偽反例を除くように抽象モデルを修正する。

---

[151] 正確な条件は、Clarke, E., Grumberg, O. and Peled, D.: Model Checking, MIT press, 1999 の Theorem 16 ACTL*の健全性に示されている。ここでは、直感的な表現で条件を示している。

[152] 状態遷移システムに関する等価性の関係として双模擬関係(Bisimular relation)がある。この関係が成立つ状態遷移システムは、見かけ上区別できないため、抽象化の健全性も成立つ。

[153] 抽象モデルと具体モデルが双模擬関係にあるとき、健全性と反例の保存性の両方が成立つ。Blackburn 他編, Handbook of Modal Logic, Theorem 4.3

モデル検査における抽象モデルと具体モデルの関係は以下のように分類される。

表 9-1:抽象モデルと具体モデルの関係と用途

分類		説明	用途
(1)	(A) 健全性	抽象モデルで検証した性質は、具体モデルでも成立つことが言える。	性質の保証に有用
(2)	(B)反例の保存性	抽象モデルで反例がある場合、具体モデルでも反例が存在する。	不具合検出に有用
(3)	(A)かつ(B)	—	上記両方に有用
(4)	(A)または(B)が経験的に成立つと分かる	—	実際この場合が多い

分類(1) は、抽象モデルである性質が検証されれば、元の具体モデルでも成立つことが保証されるため有用ではあるが、実際には、健全性を示すことが難しい場合が多い。また、健全性の条件を満たすためには、状態数をあまり減らすことができず、状態爆発の回避という観点で効果が低い場合が多い。分類(2)は、抽象モデルで反例が見つかった場合、具体モデルでも反例があることが示される。これも実際には、抽象モデルで反例が存在しても、具体モデルでは性質が満たされる場合があり、そのような反例を「偽反例」と呼ぶ。

分類(3)は、検証性質に関して、抽象モデルと具体モデルが同じであることを示しており、非常に強い関係にあるが、実際に適用できるのは稀である。現実的には、(4)の場合が多く、アドホックに検証に関係ない部分を捨象する方法が使われることが多い。この場合、抽象モデルの検証結果と、具体モデルの性質が厳密には保証されないが、その場合でも反例発見やソフトウェアの品質に対する確信度を向上させるという点で効果が得られる。

9.3. 抽象化および状態爆発の対策の全体像

モデリングのプロセスに従い、抽象化と状態爆発の対策法について全体像を示したものが図 9-5 である。

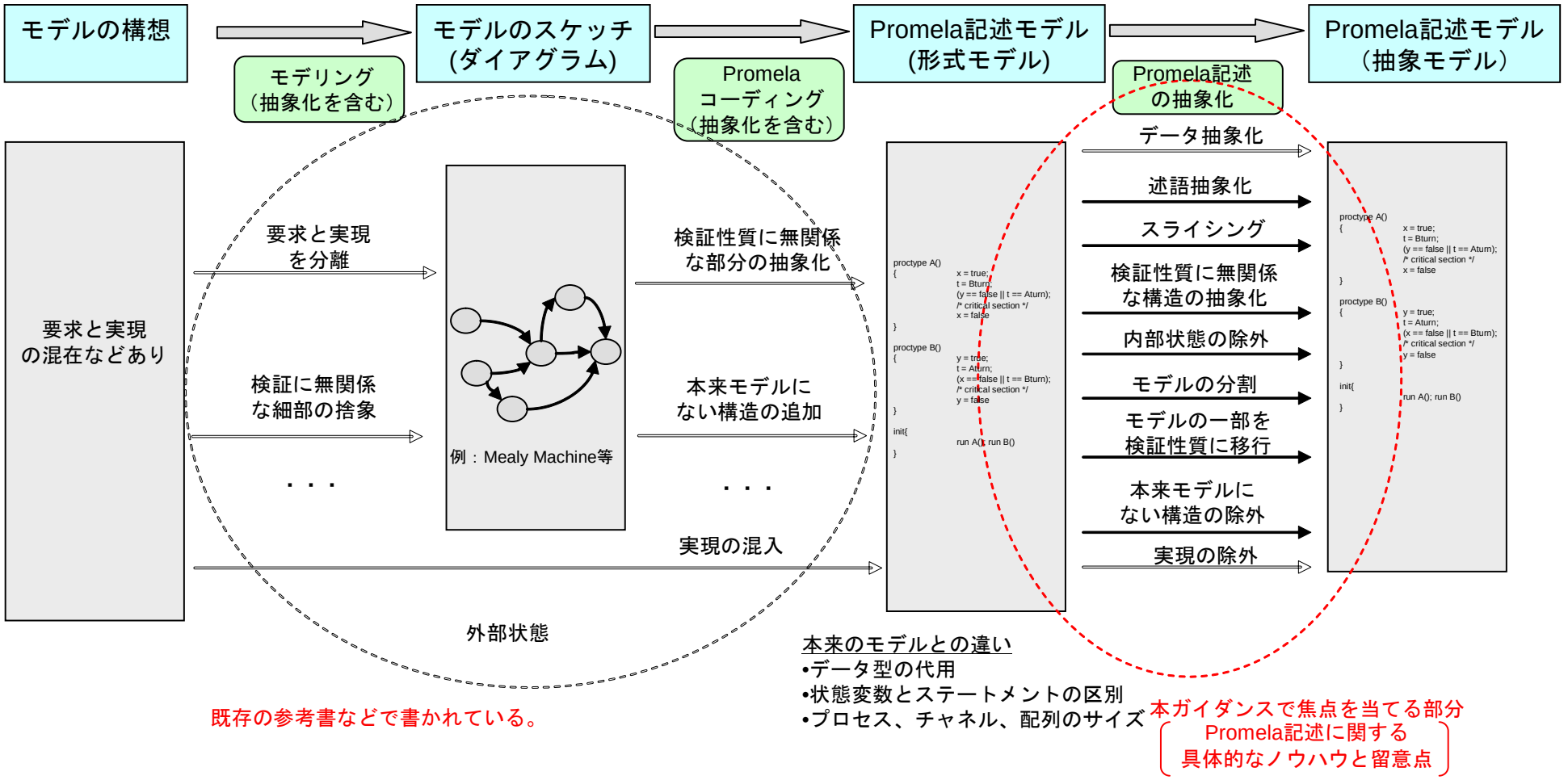

図 9-5:モデルの抽象化と状態回避策に関する全体像

抽象化のプロセスは、モデルの構想から始まり、ステートマシン図などのダイアグラムを用いてモデルのスケッチを行い具体化させる。さらに、Promela 等のフォーマルメソッド記述言語を用いて厳密なモデルを定義する。これに対して、状態爆発等の問題が発生する場合、次節に示す対策法を用いて抽象モデルを作成する。

モデルの構想からダイアグラムによる表現の段階で、検証範囲に入らないコンポーネントを除外したり、要求と実現の分離をおこなうなどアドホックな抽象化を行う。ダイアグラムから形式モデルの記述においては、具体的なデータ型の定義やチャネル通信のセマンティスクを決定する。これにより、ダイアグラムのレベルでは曖昧であった意味を厳密に規定する。

次節では、Promela 形式モデルから抽象化された形式モデルへのプロセスにおいて、Promela のコードレベルで具体的な対策について示す。

9.4. Promela 記述に関する状態爆発の対策

この節では、Promela により記述された形式モデルに関して状態爆発を回避するためのコードレベルでの具体的な対策法を示す。

Promela の構成要素に基づき以下のレベルで対策法を示す。

(1) 並行プロセスに関する対策
　検証性質に影響を与えない並行プロセスを統合、除去することで状態爆発を回避する。
(2) 変数に関する対策
　不要な変数や変数の使い方の不注意による状態爆発を回避する。
(3) ステートメントに関する対策
　内部状態の抽象化などにより状態爆発を回避する。
(4) モジュール分割等に関する対策
　検証に無関係なモジュールの除外などにより状態爆発を回避する。

9.4.1. 並行プロセスに関する対策

9.4.1.1. 問題の原因

第9.1節に示した並行プロセスの非同期実行により、各プロセスの実行ステートメントがインターリービングされ、プロセス数の組み合わせのオーダーで状態爆発を発生する。一般に、状態数 N のプロセスと状態数 M のプロセスが非同期平行動作すると、N×M のオーダーの状態からなる積状態遷移システムができる。並行プロセスが増えると、プロセス数の指数オーダーで爆発する。このようなことから、プロセス数の削減は状態爆発抑止に非常に効果が大きいことが分かる。

9.4.1.2. 対策

9.4.1.2.1. メッセージ送受信が主体のプロセスを統合

メッセージの送受のみや単純なフィルタのみを行うプロセスを、複数のプロセスで行うことがよくあるが、これらは必ずしも検証上は別プロセスにする必要がない場合が多い[154]。文献に書かれている例を参照するとよい。メッセージの送受に複数のプロセスを利用している場合、まずこれをより簡潔に記述できないかを考えてみることは検討に値する。

ブルーレイディスクのケーススタディの場合、アプリケーションに対して、ディスク操作要求を任意の順序で、任意回実行するユーザプロセスは、その要求を受け取るアプリケーションのプロセスと統合できる。

9.4.1.2.2. 変数値管理のプロセスを除去

オブジェクト指向モデリングの影響などをうけて、1 つあるいは複数の変数値を管理するだけのプロセスをモデルに含める場合がある。しかし、モデル検査の立場から言うと、これは避けた方が良い。このような場合は、通常、プロセス間で共有されるグローバル変数を用意して、それを、それぞれのプロセスが必要な時点でアクセスする形でモデル化する方が不要な状態を増やさずに済むのが普通である。

9.4.1.2.3. 同期通信型のプロセスの統合

シーケンス図で書いたときに「縦線」になるものをそれぞれ別個のプロセスとしてモデル化してしまう場合がある。例えば図 9-6 のようなシーケンス図がある場合、A は B の返却値があるまで実質的に動作を停止し、B は C の返却値があるまで実質的に動作停止するということであれば、A, B, C をそれぞれ別プロセスにする必要はなく、全体を 1 つの逐次的なプロセスとすればよい。

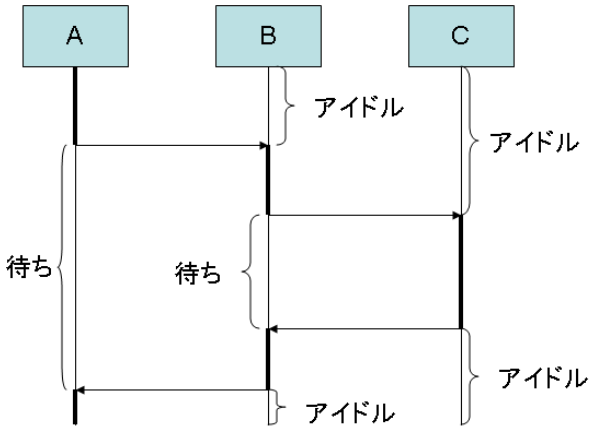

図 9-6:実質的に逐次的な処理のシーケンス図

[154] The SPIN MODEL CHECKER, 第 5 章“Sink, Sources, and Filters”

これに関連して、特に SPIN/Promela の場合に、慣れないうちは、単なるサブルーチンを独立プロセスにしてしまう場合があるが、これは Inline 定義を利用すべきである(なお inline 定義については必要な記述上の注意などもあるので例えば文献の inline 定義の注意を参照のこと)。

9.4.1.2.4. 検証性質に基づくプロセスの統合

これは上述の(1) から (3) のまとめであるが、これら以外にも、検証項目(検証式で表されるもの)との関係では関係ない部分は 1 つのプロセスと考えてしまうことができる場合もある。場合によっては、検証式自体を書きかえることで、そのことが可能になる場合もある。

9.4.1.2.5. 多重プロセスの試行的調整

この方法は、上述の(1)から(4)とは異なり、主として不具合発見の目的でのみ有効な方法である。しかし、実際の検証ではモデル上の変更量が小さくて済み、またもとのモデルに戻すのも簡単であることから、しばしば利用される。例えば、顧客が N 人いて、それぞれ別個にシステムにアクセスしてくる、という状況をモデル化するとき、それぞれの顧客を別プロセスにする必要があるとする。このとき、小さい N で示すべき性質が成立するかをまず試してみて、次第に N を必要な大きさまで大きくしていく、という手法をとることができる。

通常、N が m のときに反例がみつかったとすると、N を m+1 にしたときもその反例は有効である。特に最後に追加された m+1 個目のプロセスが完全にインターリーブで動くとすると、たまたまそのプロセスが動く順番がまわってこない場合を考えられるが、これは N が m のときの動作とほぼ同一と考えられる。したがって、多くの場合、N が m のときの反例は、そのまま N が m+1 のときの反例にもなる。しかし逆に N が m のときに反例が出なかったからといって N が m+1 の時にも反例が出ないことにはならないのが通常である。新しく加わったプロセスのせいで別の処理の組合せが発生するので、そこで反例が発生する可能性があるからである。
このことから、この N を小さくする方法は、特に、不具合発見の目的に有効な方法であるということになる。

ただ、ここで注意しなければいけないことは、反例についても、N を大きくしたときに必ずそのまま保存されるとは限らないということである。一般にこの手法を使うときには、小さな N で反例が出たら、反例が出た原因を調べて、大きな N ではどうなるかを推定することが必要である。

9.4.2. 変数に関する対策

9.4.2.1. 問題の原因

変数ごとに取る値のサイズを掛け合わせたオーダーで状態数が増加する。したがって変数の数の指数オーダーで急激に増加する。

たとえば、Promela で、以下の処理を考える。

```
bool flag;
active proctype pr1() {
  ...
```

このとき、pr1 の中に記述された表面的なコードは同一でも、モデル検査系が調べる状態は、大域変数 flag が true のときと false のときで別の物である。したがって、flag の値が pr1 の中で両方の値をとりうるならば、Promela コード上に表面的に見える状態数の 2 倍の状態をモデル検査器は調べる必要がある。

ここでもし変数 flag が不用ならば、それを消してしまえば、別の物と扱われていた 2 つの状態は縮退して 1 つになる。これによって、大まかに、状態数は半減することになる(実際には pr1 の中での flag の使い方にも依存する)。

基本的に変域の大きさ R の変数が V 個あるときには、モデル検査器内部には R×V 個の状態が生じることがあると考えるのが良い。従って両者をなるべく減らすべきである。表 9-2 に Promela の場合の型と変域の大きさの例を示す。他のモデル検査系を利用する場合も同様に考えることができる。

表 9-2:型と変域の大きさ

Promela の型(あるいは変数宣言)	変域の大きさ
mtype = { red , blue, yellow }	3 + 1 (例外値)
bool	2(True or False)
Boolean a[5]	2 の 5 乗=32
byte	2 の 8 乗=256
short	2 の 16 乗=65536
Int	2 の 32 乗≒43 億

Typedef Record {
byte b[3]
short f
}
	256 の 3 乗×65536≒280 兆

9.4.2.2. 対策

9.4.2.2.1. 必要最小限のサイズの型を用いる

Promela なら、なにげなく int 型を使っている部分があったら、mtype や, byte, bit で十分ではないか、あるいはせめて short ではどうかを検討することは有効である。

9.4.2.2.2. 構造型の定義はより単純な構造を用いる

typedef で定義されたレコード型などを利用している場合、表 9-2 からわかるように、全体の変

域のサイズは各フィールドの変域のサイズの積となる。従って、不要なフィールドを削ることを考えること、あるいはそのフィールドの型をよりシンプルなものにできないかを検討することも有効である。このことは、その型を利用する変数の変域を小さくすることに貢献する。

9.4.2.2.3. 検証に関係ない状態変数は削減する。

実装に整合させることを意識して、不要な変数を入れてしまうことは考えられる。しかし、それが検証に関係がなければ、入れる必要はない。

なお、変数が関係あるかないか正確に判定するには影響範囲をたどっていくCOI(Cone of Influence)と呼ばれる方法があるが、まずは、明らかに関係ないものを排除することを考えるべきである。なお、COIについて詳しくは専門的になるが文献[155]の 13.1 節を参照すると良い。

また、使用する変数との関連を考慮して、検証項目自体を変形する(分割するなど)ことも検討対象となり得る。

9.4.2.2.4. グローバル変数をプロセスローカル変数にする。

検証に関係の無い大域変数を局所変数にすることで、局所変数はスコープの外で使われないことが分かるため、SPINの処理系で最適化される[156]。

9.4.2.2.5. モニタリング用変数、デバッグ用の変数は削除する。

通常のデバッグ時に状態をモニターする変数を挿入する感覚で、モニタリング用の変数を挿入している例があるが、このような変数は状態数を不要に増大させる可能性があるので、避けるべきである。

文献[157]第 5 章の"COUNTERS"には、このような例として実行ステップを数えるカウンターを除去する例が説明されている。

9.4.2.2.6. データ抽象化

データの変域を小さくする抽象化法としてデータ抽象化（データマッピング）が代表的である。これは、例えば、整数として宣言された変数が、検証においては、マイナス、0、プラスに注目すれば十分である場合など、整数をこれらの3つの値に縮退させる方法である。

データ抽象化については以下のような文献で豊富に説明があるため、ここではそれらへのポインタを示す。代表的な参考書に以下のようなものがある:

- 文献[158]の 9.2.1 節
- 文献[159]の 2.3 節および 2.6 節

---

[155] Clarke, E., Grumberg, O. and Peled, D.: Model Checking, MIT press, 1999.
[156] 萩谷, 吉岡, 青木, 田口, SPIN による設計モデル検証, 2008
[157] Holzmann, G.:THE SPIN MODEL CHECKER, Addison Wesley, 2004.
[158] 中島震: SPIN モデル検査 -検証モデリング技法, 近代科学社, 2008 .
[159] 田辺、高井、高橋: 抽象化を用いた検証ツール, コンピュータソフトウェア, Vol.22, No.1,pp.2-44,2005.

さらに、データ抽象化のいろいろな種類については、文献 の 9.2 節に詳しい。

9.4.2.2.7. 述語抽象化

データ抽象化を拡張したより専門的な手法として、述語抽象化という手法がある。述語抽象化は、複数の変数の組合せに対して、それらを単純な構造に抽象化するものである。これらについても文献 、などに書かれているため参照すると良い。

9.4.2.2.8. キュー、スタックのサイズは小さいものから試す

これは上述の 9.4.1.2.5 節と同様に、主として不具合発見の目的の際のみに使用できる方法である。キューやスタックが配列として実装されている場合、その配列のサイズを小さく設定してとりあえず反例が出ないかどうかを確認し、次第に本来のサイズまで広げていくという方法である。これも 9.4.1.2.5 節と同様に、キュー、スタックなどを小さくして検証した結果反例が出たからと言って、大きい場合でも反例が出るとは一般には言えない。反例の出た原因から推定することが必要になる。

なお、例えばキューに関して、9.4.2.2.6 節のデータ抽象化の応用として、実際のキューを作らずに、例えば「キューに空きがあるか満杯か」だけを表す二値変数だけを用意することでも検証目的によっては十分な場合がある。実際にこの方法でデッドロックを検出した例も存在する。本手法は 9.4.2.2.1 節から 9.4.2.2.6 節と比較すると最後の手段であるので、9.4.2.2.1 節から 9.4.2.2.6 節までを先に検討するとよい。

- 注意点

1. ローカル変数への着目

変数を削減する、あるいは変数の変域を削減することを検討する際に、ローカル変数をまず見直してみることは有効である。

ローカル変数は、モデル記述上一つであっても、実行時にそれぞれの実行主体ごとに別空間を利用する場合があるので、実際の数は増幅する場合があるからである。ローカル変数がどのようにインスタンシエートされるかを正しく理解した上でその削減を検討できれば一番ではあるが、そうではなくても削減対象としてローカル変数に着目することは有効である。

2. 変数削減時の処理の変形方法

上記(2-3)のように変数を削減しようとすると、その変数を参照して処理を変えている部分をどのように変形すれば良いかが問題になる。 例えば、Promela 記述内に次のような部分があるとする。

URL: < http://www.jstage.jst.go.jp/article/jssst/22/1/1_2/_pdf/-char/ja >.

```
if
  :: flag = true -> ACTION1;
  :: flag = false -> ACTION2;
```

ただし、ACTION1, ACTION2 は実際にはそれぞれ何らかの処理であるとする。 このとき、flag が全体から見ると検証には関係がないので削除するということになったとすると、これをどのように変形すべきか、という問題になる。

この一般的な方法は、次のように ACTON1 または ACTION2 への非決定的な遷移にする方法である。

```
do
  :: ACTION1;
  :: ACTION2;
```

この方法で反例がみつかったからといって、それがもとのモデルでの反例になるとは限らない。もちろん本当に flag が検証項目と関係がないならば、反例にも関係ないはずであるが、一応、この方法で発見された反例がある場合は、それを解析してもとのモデルでも反例となるかを調べる必要がある。

9.4.3. ステートメントに関する対策

9.4.3.1. 問題の原因

Promela で記述したモデルでは、1 つの文が1つの状態を構成する。複数の文が 1 つのステップとしてアトミックに実行されたと見なしても問題が無い場合、複数の文を atomic を用いて1ステップと定義した方が、他のプロセスのステップとの間のインターリービングが発生しない分状態を抑えることができる。

処理の遷移の途中に中間状態があり、本来 1 ステップで良いところが 2 ステップになっているとする。そして、そのプロセス以外にも他にいくつものプロセスがあるとする。すると、中間状態があるがゆえに、インターリーブの考え方のもとで他のプロセスとの間の相互作用の組合せが膨大に増えることになり、結果的に、モデル検査対象の経路数が巨大化する。

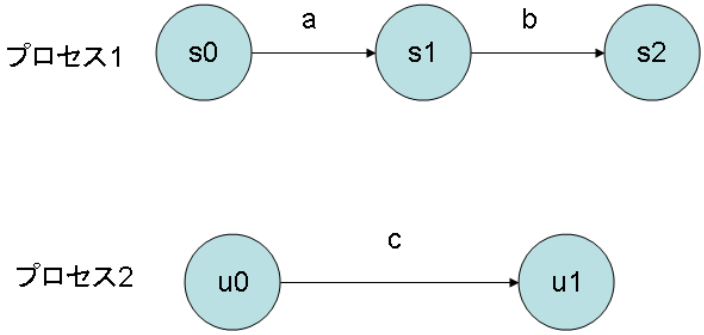

図 9-7:冗長な中間状態 s1 がある場合

例えば、図 9-7 のように 2 つのプロセスがあるとする。このとき、2 つのプロセス内のアクション a,b,c のインターリーブによる実行順の組合せは、

- c; a; b
- a; c; b
- a; b; c

の三種類である。しかし、もしプロセス 1 の中間状態 s1 が冗長であるとすると、これを削除して、a と b をまとめて一つのアクションとしてしまうことが考えられる(図 9-8)。

すると、a,b,c の可能な実行順の組合せは、

- c; a; b
- a; b; c

の二種類に減少する。これに応じて、モデル検査器の探索空間も小さくなる。

この例では他のプロセスは 1 つだけであり、アクションも 1 つだけを考えたが、通常は他にたくさんのプロセスがあり、いくつものアクションが関係してくるので、1 つの中間状態を削除しただけでも多くの余分な組合せの排除につながる。

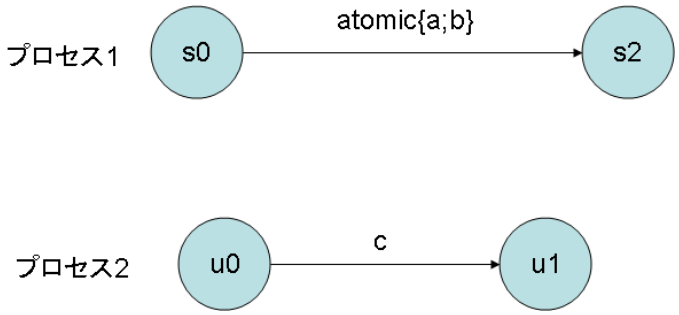

図 9-8:状態 s1 を省いて、a;b を 1 ステップにした場合

9.4.3.2. 対策

9.4.3.2.1. 内部状態の遷移系列を1ステップとなるように atomic宣言する。

Promela の場合、まとめて 1 ステップとしたい処理を atomic{}で囲うことでこれを実現できる。これにより劇的な効果をもたらす場合がある。ただし、とにかく atomic をつけると状態爆発に効果が出るということを知ると、atomic をつけてはいけない部分にまで付けてしまうことがある。1 ステ

ップにするということは、その処理が始まって終わるまでの間は他のプロセスが動けないということである。atomic をつける際に、そこで他のプロセスが動く場合を考えなくてよいかどうかを考慮する必要がある。

ここまでの説明は SPIN/Promela に沿って行ったが、それ以外のモデル検査ツールを利用する場合も基本的な考え方は同様である。

9.4.3.2.2. スライシング

ステートメント単位で、注目したスライス基準の依存関係を辿り、無関係なステートメントを除外する。スライシングについては文献[160]に具体的に書かれているので、それを参照するとよい。
また、SPINでは自動スライシングの機能がありオプション機能(-A)を指定することで利用できる。

9.4.4. モジュール分割に関する対策

9.4.4.1. 問題の原因

モデル全体に対して複数の検証性質を順に検証しようとすると、モデルの状態数が大きすぎる場合が生じる。検証性質ごとに、必要最小限のモデルに分割して検証することが可能である。

9.4.4.2. 対策

9.4.4.2.1. モジュール分割

モジュール分割は、モデリング言語の種類によっては、強力な サポートがある。Promela の場合には特に言語的なサポートはないが、処理を意味的な関連度の高さでグルーピングした後、分割するのが適当である。

9.4.4.2.2. 初期状態による分割

初期状態がいくつもある場合、全体の状態の中には、特定の初期状態からしか辿りつかないものが含まれている可能性がある。このため、初期状態ごとに関係がある状態だけをモデル化することも効果がある場合がある。

また本質と関係ない初期値の未定値(例えば配列でキューやスタックを表現するときの、値の格納前の配列の値など)は確定しておくことで、不要な初期値についての計算を避けることができる。例えば、図 9-9 が左から使用するスタックを表すとして、スタックポインタより右の部分の値(図の e1 から e7)は不定値の可能性がある。これを確定しておかないと、モデル検査器は e1 から e7 に入りうるすべての値の組合せについて調べてしまう場合がある。したがって、スタックの空の部分の初期値は何らかの値に確定しておくべきである。

[160]磯部，粂野，櫻庭，田口，田原：ソフトウェア科学基礎，近代科学社，2008

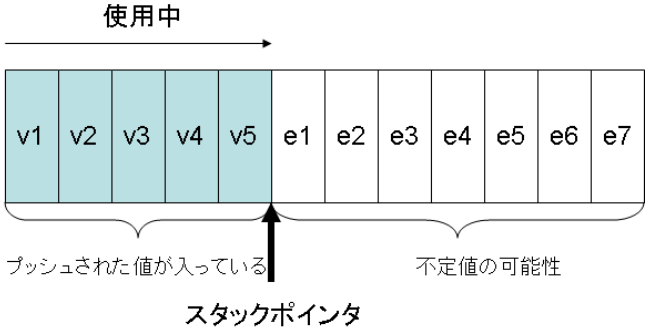

図 9-9:スタックの利用前部分の不定値

9.4.4.2.3. 検証項目による分割

検証項目が複数ある場合、それごとに、9.4.1 節～9.4.3 節の手法の適用可能性が変化する場合があるので、それぞれ別個にモデルを作って検証することも効果がある。

9.4.4.2.4. 例外的処理の分離

システムは通常、標準的な処理と例外的な処理を含んでいる。これをすべて一体化して検証しようとすると状態数が増大することになるのが一般的である。不具合発見が目的であれば、例えばまず例外処理部分を除いたモデルで標準的な処理について検証(不具合調査)し、その後、例外処理を含むモデルを検証することも考えられる。あるいは、例外処理への脱出までのモデルと、例外処理自体のモデルを別に作り、それぞれ別個に検証する方法も考えられる。

9.4.4.2.5. 仮定を利用した検証条件の分割

性質 B を示すのに、まず A を示し、次に「AならばB」を示すことで、結果的にBを示すという方法を使うこともできる。これも一種の分割と言えるであろう。

一部の性質は、モデル化するよりも、検証条件に仮定(⇒の前提)として与えた方が扱いやすい場合がある。例えば、公平性と呼ばれる性質(インターリーブで考えたときに、どのプロセスにもいつかは動く順番が回ってくる、という性質など)はモデル上には表現しにくい場合が多いので、これを仮定にするということはよく行われる。

ただし、検証性質にモデルの性質を入れた場合、モデル検査の状態数が大きくなるため、これは状態爆発に対する対策というよりも、モデルの記述の困難性に対する対策と言える。

- **適用上の注意点**

上記の9.4.4.2.1節から9.4.4.2.5節のいずれについてもあてはまるが、分割検証できるようにするために、検証項目の記述自体を工夫する(分割するなど)も考える余地がある。通常細かい検証項目ほど、ローカルに検証できるので、検証すべき項目を細かく詳細化することも一つの方法である。

9.5. 本章のまとめ

本章では、状態爆発の問題とその対策法としてモデルの抽象化のうち、Promela のコードレベルでの方法を中心にまとめた。データ抽象化や述語抽象化など、一般的な手法に関しては、既存の文献に具体的に書かれているため、それらの紹介にとどめた。

本章により状態爆発の問題その解決策としてモデルの抽象化に関する全体像を把握するとともに、問題の原因の理解を通じた、実践での解決のためのヒントを示した。