社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－SE－159 (6)
2008／3／17

モデル検査のためのアスペクト指向でのモデル記述支援環境

大野 真一朗[†1] 岸 知二[†1]

本稿では、モデル検査器 SPIN を用いた検証における仕様記述言語 promela でのモデル記述をアスペクト指向技術を用いて支援するための、アスペクト指向言語、言語処理系を提案する。一般に、promela での検証モデルは、検証目的に基づき記述されるため検証目的が変わるとそれに伴い検証モデルにも変更が必要になる。このモデルの変更にアスペクトを用いることにより様々な検証を効率的に行う方式を実現する。

Aspect oriented model description environment for model checking

Shinichiro Ohno [†1] and Tomoji Kishi[†1]

In this paper, we propose an aspect oriented language and language processing system, called Weaver, for Spin model checker. As verification model is defined depending on verification purpose, we have to re-write the model if we change the verification purpose.

Utilizing our language, we can encapsulate these changes apart from the verification model, and effectively develop verification model.

1. はじめに

近年、ソフトウェアは金融、公共、産業、交通システムから自動車、家電など至る所で利用されている。そのため、そのソフトウェアの誤りが、社会与える影響は大きくソフトウェアの信頼性向上が大きな課題となっている。そうした背景の中、ソフトウェアの正しさを確かめる手法の一つであるモデル検査[2)]が、ソフトウェア検証の新たな選択肢として注目されている。

モデル検査は、検証対象の有限状態モデルと、そのモデル上で満たしたい論理的性質を与えると、モデル上でその性質が成り立つかを自動的に検証できる技術であり、そうしたモデル検査を支援するツールの一つに SPIN[1)] がある。

1.1 検証モデルの変化

SPIN で検証を行うには、仕様記述言語 promela で検証対象をモデル化し、SPIN を用いて検証器を生成し、その検証器を実行することで、仕様の正しさを確かめる。この promela による検証モデルは、一般に検証目的に応じて異なったものとなる。例えば、検証目的が「ある機能Aに関わる性質の検証」の場合には、「機能Aに注目したモデル」を作成する必要がある。次に、検証目的が「ある機能Bに関わる性質の検証」に変化した場合には、先ほどのモデルを「機能Bに注目したモデル」へと書き換える必要がある。

1.2 横断的なモデルの変更

検証モデルの変化の典型例として、「アサーションの埋め込み位置の変化」が挙げられる。アサーションは、モデル中に論理式と共に記述され、その記述された場所で論理式が成立するかを検査する文である。このアサーションを用いてモデル中のある機能Aの事後状態を確認したい場合、モデル中の機能Aに関する記述の事後にアサーションを埋め込む必要がある。一方、このモデルに対して機能Bの事後状態を確認したい場合には、先ほどのアサーションを全て取り除き機能Bの記述の事後にアサーションを埋め込む必要がある。

この様に、モデルに横断的に現れるような「仕組み」や「機能」に関わる性質の検証を行う場合、検証目的に応じたモデルの変更は煩雑であり、検証支援系による支援が望まれる。

1.3 アプローチ

我々は、仕様記述言語 promela 向けのアスペクト指向言語を設計し、その言語処理系を開発するとともに、この言語を用いて先の問題を解決する方法を提案する。これは、検証対象を表すモデルを一つ作成し、検証目的に対応するアスペクトをそれぞれ準備することにより、検証目的の変化によるモデルの変更をアス

†1 北陸先端科学技術大学院大学 情報科学研究科
School of Information Science, Japan Advanced Institute of Science and Technology

ペクトの変更により実現するというものである。

1.4 提案する言語

本稿で提案するアスペクト指向言語は、代表的なアスペクト指向言語の一つである Aspect/J[4] のジョインポイントモデル（ジョインポイント、ポイントカット、アドバイス）を promela 向けに拡張し実現する。

1.5 本稿の構成

2 節では、提案するアスペクト指向言語の要件とその実現方法について述べる。3 節では、提案言語の文法と記述例について述べる。4 節では、言語処理系の実装について述べる。5 節では、適用例とその結果について述べる。そして、6 節でまとめと今後の課題について述べる。

2. 提案言語の要件とその実現

本稿で提案するアスペクト指向言語を設計するにあたり、promela の言語としての特徴や Aspect/J の対象言語である java との相違点を検討し、提案言語が持つべき機能を要件とし以下の 2 点が重要であると判断した。

(1) チャネル通信に関する言語要素を指定する文法が高い表現力を持つこと。
(2) proctype 中の任意の範囲を指定でき、その範囲に対してアスペクトを作用させることが出来るような機構を持つこと。

2.1 要 件 1

2.1.1 promela でのチャネル

promela は本来、プロトコル検証モデルを記述するための仕様記述言語であり、プロトコル検証におけるメッセージ伝送路のモデル化には、チャネルが用いられる[3]。また、ソフトウェア検証においてはプロセス間通信や同期といった並行動作するプロセスが持つ相互の関係をモデル化ために、チャネルが用いられる。プロトコル検証におけるメッセージ伝送路、ソフトウェア検証におけるプロセス間通信、同期の機構はいずれも検証モデルの重要な要素である。そのため、検証時にこれらの要素に関心があることが多く、アドバイスを記述する際にもチャネル操作にアスペクトを作用させたいことが多いと考えられる。よって、様々なチャネル操作にアスペクトを作用できるように、チャネル操作のジョインポイントを指定するための文法は高い表現力を持つ必要があると考えた。

2.2 要件 1 の実現

promela でのチャネル操作文は、チャネル名、操作、メッセージ名の 3 つから構成されるため、チャネル名、メッセージ名に関してはこれを正規表現で指定できるようにするとともに、操作に関しては全ての種類の操作を指定できるように文法を設計し、様々なチャネル操作に対してアスペクトを作用させることが出来るようにした。

2.3 要 件 2

2.3.1 Aspect/J でのジョインポイントの指定

Aspect/J では、ジョインポイントとしては、「メソッド、コンストラクタの呼び出し位置や実行位置」、「フィールド参照位置や代入位置」、「インスタンスの初期化位置」等があり、ポイントカットとしては、「メソッドパターン（クラス名やメソッドのシグニチャなど）」、「コンストラクタパターン（修飾子、クラス名、引数など）」、「フィールドパターン（クラス名、修飾子、フィールドの型など）」等がある。

Aspect/J ではどのようにしてポイントカットを記述するかを図形描画アプリケーションを例として考える。このアプリケーションは、次の 3 つの処理から構成されるとする。

(1) 描画する画面の初期化
(2) 図形の描画
(3) 描画した図形の削除

1 の初期化を実行した後に、アスペクトを割り込ませたい場合は、「初期化を行うメソッドの実行」をポイントカットにより指定する。そして、after アドバイスを用いることによりこれを実現する。java では、「メソッドが一つの意味上の単位」となっているため、「初期化メソッドの実行」をポイントカットで指定することにより、コード上で初期化を行っている部分の事後を指定することが出来る。

2.3.2 promela の言語要素

一方、promela には、メソッドなどのコード上の範囲に対して意味を与えられるような機構が無い。inline 関数やマクロなどは関数に似た機能を提供するが、これらはコードの置換によってこの機能を実現しているため、プリプロセス後は関数としての意味を持たない。そのため、先ほどの描画アプリケーションの例のようなポイントカットを promela のモデルに対して記述する場合、「初期化を行っているコード上の範囲」を promela の言語要素を用いて指定することが難しい。

promela に対して Aspect/J のジョインポイントモデルを適用する場合には、この問題を解決する必要があり、要件 2 を満たす必要があると考えた。

2.4 要件 2 の実現

Java でのメソッドのように「あるコード上の範囲」を指定するために、promela の言語要素を「範囲を持つもの」と「持たないもの」に分類し、「範囲を持つ

言語要素」に対しては、これを用いて promela のコード上の範囲を指定する方法を考案した。一方、「範囲を持つ言語要素」に対しては、そのポイントカットでの指定に対して「2つの意味」を与え、それらを用いてアスペクトの作用する範囲を適切に絞り込む方法を考案した。これらの詳細について、以下に述べる。

2.4.1 言語要素の分類

要件 2 を実現するための方式を考案する際に、promela の言語要素を表 1 のように分類した。

表 1 言語要素の分類

分類	言語要素の BNF での名前
範囲を持つもの	if, do, atomic, d_step, unless, option, init, never, trace, notrace, proctype, label
範囲を持たないもの	send, receive, xr, xs, goto, print, assert, expr, one_decl, assign

例えば、proctype 等はプロセス名と言語要素名を指定することにより、コード上の範囲を指定することができる。しかし、send（チャネルの送信文）等は内部に文を含むことはできない。この様に内部に文を含むことが出来るものを「範囲を持つもの」とし、それ以外を「範囲を持たないもの」として分類した。

2.4.2 範囲の指定方法

a) 言語要素による範囲の指定

「範囲を持つ言語要素」に対しては、「その言語要素が囲う範囲を指定したこととする」という意味をこの指定に与える。これにより「範囲を持つ言語要素」を用いてコード上の範囲を指定することが出来る。

b) 範囲への名前付け

図 1 の例を考えると、「範囲を持つ言語要素」による指定では、すべての if が囲う範囲を指定することはできるが、二つ目の if が囲う範囲といった特定の範囲を指定することは出来ない。このように、言語要素名のみの指定では、特定の範囲のみを指定することができない。そこでラベルを用いて特定の範囲を指定する方法を導入する。

ラベルは文 (BNF での stmnt) に付加することにより、その文に名前を付けることが出来る文である。このラベルを「範囲を持つ言語要素」に付加することにより、その言語要素が囲う範囲に対して名前を付けることが出来る。

たとえば、図 1 では、if 文に対してラベル「L1」を付加している。このモデルに対して単に if 文を指定すると、if 文は二つとも指定されたことになるが、「L1」というラベルが付加された if 文という様に指定することにより 2 つ目の if 文のみを指定することが出来る。

c) 任意の範囲の指定

b) では範囲を持つ言語要素にラベルを付加することにより、範囲を指定したが、図 2 ようなモデルは範囲を持つ言語要素がないため、この方法では範囲を指定することが出来ない。

このようなモデル中で、ある範囲に名前を付けたい場合には BNF の stmnt の中の中括弧を用いる。中括弧で BNF の sequence を囲うとその中括弧自体が BNF 上では stmnt となるためラベルを付けることが出来る。図 2 において任意の範囲に対してラベルを付加したい際には、図 3 のようにその範囲を中括弧で囲みラベルを付加すればよい。

図 1 if 文にラベルを付加した例

```
if
  ::skip
fi;
//[L1]+[if]の範囲はここから
L1:if
  ::skip
fi
//ここまで
```

図 2 範囲を持つ言語要素が無い例

```
ch!init;
printf("init\n");
ch?ack;
ch!data(1,2);
printf("send data\n");
ch?ack;
```

図 3 中括弧で囲った例

```
INIT:{
    ch!init;
    printf("init\n");
    ch?ack;
    }
SEND:{
    ch!data(1,2);
    printf("send data\n");
    ch?ack;
    }
```

2.4.3 範囲指定の二つの意味

これまで、「範囲を持つ言語要素」の指定に対して1つの意味を与えていたが、ここで、それを拡張し「範囲を持つ言語要素」の指定に対して次の2つの意味を与える。

(1) 範囲を持つ言語要素を指定すると、その範囲自体をブロックとして指定する。(今までの意味)

(2) 範囲を持つ言語要素を指定すると、その範囲中の全てのジョインポイントを指定する。(拡張した意味)

この2つの違いを説明するために、atomic という言語要素に対して、/*aspect*/というコメントを事後に割り込ませる例を図4に示す。

意味1で範囲を考えるとatomicで囲われた範囲をブロックとして扱うので、そのブロックの事後、つまりatomicの中括弧の後にコメントが追加される。(図4の右上)

意味2で範囲を考えるとatomicで囲われた範囲中の全てのジョインポイントが指定されるので、各skipの事後にコメントが追加される。(図4の右下)

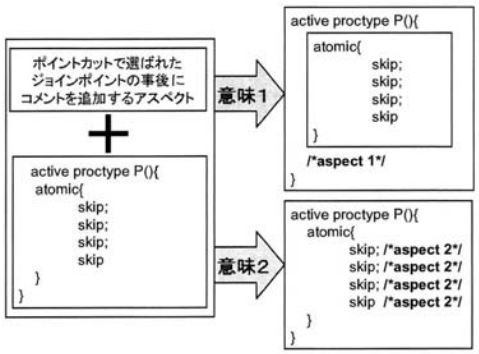

図4 範囲の意味の違いによるウィーブの違い

この2種類の意味は以下のように活用することができる。

・意 味 1

「範囲を持つ言語要素」を意味1で扱うと、範囲自体に対して操作を行うことができる。つまり、ポイントカットで指定されたif,do,option等のブロックとしての書き換え、事前・事後へコードの追加ができる。

・意 味 2

「範囲を持つ言語要素」を意味2で扱うと、その言語要素が囲う範囲の全てのジョインポイントを指定できるため、「範囲の指定＝ジョインポイントの集合」として考えることが出来る。このジョインポイントの集合に対して、和集合、積集合などをとることによりアスペクトが作用する範囲を適切に指定することが出来る。

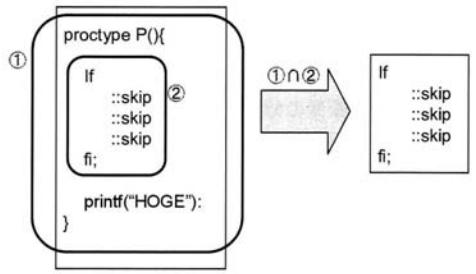

図5 ジョインポイントの集合に対する演算

例えば図5の「① プロセス名Pのproctypeが囲う範囲の全てのジョインポイント」と「② ifが囲う範囲の全てのジョインポイント」の積集合をとると「①∩② プロセスP内のif文が囲う範囲の全てのジョインポイント」を指定することが出来る。このようにしてproctype中の範囲を絞りこみ、適切なジョインポイントにアスペクトを作用させる。

2.5 提案言語での範囲の扱い

これまで、「範囲を持つ言語要素」を指定した場合の「範囲」の扱いについて述べたが、提案言語では「意味1」、「意味2」の両方を利用できるようにする。文法上では、「範囲を持つ言語要素」をポイントカットを用いて指定した場合は、「意味1」として解釈されるが、演算子allStmntを用いた場合は、「意味2」として解釈されるというものである。詳しくは、3節にて説明する。

3. 提案言語の文法

本稿で提案する言語はAspect/Jのジョインポイントモデルを拡張したもので、アドバイス、ポイントカット、ジョインポイントによって構成される。本節では、まずジョインポイントについて述べ、次にアドバイスについて述べる。そして、ポイントカットと演算子について述べ。最後に記述例を用いてアスペクトの記述方法について述べる。

3.1 ジョインポイント

ジョインポイントはpromelaの文法上正しくコードを割り込ませることが出来る点である必要がある。promelaの実行は、任意の文*1を「実行可能」か「ブ

*1 任意の式 (expression) ではない。たとえば true && false && true を評価するとき、true を評価して true まで実行が進むということはない、(true && false && true) という各式全体を式文 (expression statement) として評価を行い、全

ロックされる」のどちらかに評価し、実行を進める。つまり、実行の単位は文 (BNF の stmnt) であり、「文 (stmnt) の実行」がジョインポイントとなる。

3.2 アドバイス

本文法におけるアドバイスは before,after,around の三種類である。各アドバイスは次のような意味を持つ。

(1) before
ジョインポイントのコードの直前に、アスペクトのコードを追加する。

(2) after
ジョインポイントのコードが直後に、アスペクトのコードを追加する。

(3) around
ジョインポイントのコードと、アスペクトのコードを入れ替える。

3.3 ポイントカット

3.3.1 範囲を持たない言語要素のポイントカット

範囲を持たない言語要素を指定するためのポイントカットは、表 2 の通りである。

例えば、chan ポイントカットを用いて chan("ch",”!”,"") と記述すると、チャネル名"ch"のチャネルに対して"送信操作"をしているチャネル操作文のジョインポイントに対してアスペクトが作用する。

表 2 範囲を持たない言語要素のポイントカット一覧

ポイントカット	意味
chan	チャネル操作文のジョインポイントを指定する。
else	else 文のジョインポイントを指定する。
break	break 文のジョインポイントを指定する。
xr	xr[*2]のジョインポイントを指定する。
xs	xs[*2]のジョインポイントを指定する。
print	印字文のジョインポイントを指定する。
assert	assert 文のジョインポイントを指定する。
expr	式文のジョインポイントを指定する。
decl	変数宣言のジョインポイントを指定する。
assign	代入文のジョインポイントを指定する。

3.3.2 範囲を持つ言語要素のポイントカット

範囲を持つ言語要素のポイントカットは表 3 の通りである。これらをポイントカットで用いると、その言語要素で囲われた範囲を意味 1 として扱う。

範囲を意味 2 として扱いたい場合は、allStmnt 演算子を用いる。

体の結果が false となるのでこの式文でブロックする。
*2 xr,xs はそれぞれ受信、送信を指定するチャネルアサーション
*3 option とは、::から始まる if,do 文の構成要素の BNF 上での名前である。

表 3 範囲を持つ言語要素のポイントカット一覧

ポイントカット	意味
if	if が囲う範囲を意味 1 として指定する。
do	do が囲う範囲を意味 1 として指定する。
atomic	atomic が囲う範囲を意味 1 として指定する。
d_step	d_step が囲う範囲を意味 1 として指定する。
option	option[*3]が囲う範囲を意味 1 として指定する。
init	init が囲う範囲を意味 1 として指定する。
never	never が囲う範囲を意味 1 として指定する。
trace	trace が囲う範囲を意味 1 として指定する。
notrace	notrace が囲う範囲を意味 1 として指定する。
unless	unless が囲う範囲を意味 1 として指定する。
proctype	proctype が囲う範囲を意味 1 として指定する。
label	label が囲う範囲を意味 1 として指定する。

3.4 演 算 子

ポイントカットの演算子を表 4 に示す。各 [pc*] にはポイントカットが入る。そして、各ポイントカットで絞り込まれたジョインポイントの集合に対して各演算が行われる。

表 4 演算子の一覧

演算子	意味
[pc1] && [pc2]	[pc1] と [pc2] のジョインポイントの積集合
[pc1] || [pc2]	[pc1] と [pc2] のジョインポイントの和集合
[pc1] contains [pc2]	[pc1] のうち [pc2] を内部に含むジョインポイントの集合
allStmnt([pc])	[pc] の中の全てのジョインポイントを追加
removeScope([pc])	[pc] の中の「範囲を持つ言語要素」を削除

3.5 記 述 例

アスペクトの記述例を図 6 に示す。この例は「プロセス P 内の、else を内部に含む option[*3]」をアドバイスのコードに置き換えるというものである。このアスペクトはまず、else を内部に含む option のジョインポイントをまず絞り込む。その次に、proctype ポイントカットを用いてプロセス名 P の proctype が囲う範囲をブロックとして取り出し、allStmnt によりその範囲中の全てのジョインポイントを絞り込む。

そして、この二つの集合に対して積集合を計算することにより、適切なジョインポイントを絞り込む。次に around アドバイスにより絞り込まれたジョインポイントのコードをアスペクトのコードに置き換えることにより、左側のモデルから右側のモデルへ変換している。

4. 言語処理系の実装

4.1 言語処理系の概要

本節では、言語処理系の概要を説明する。言語処理

系の全体図を図7に示す。本言語処理系は次の手順でpromelaのモデルに対してアスペクトを作用させる。

(1) 入力されたpromelaを言語処理系のXML内部モデルへ変換する。
(2) 入力されたアスペクトをXpath式に変換する。
(3) 2のXpath式を用いて、1のXML内部モデル中のアスペクトを作用させるジョインポイントを絞り込む。
(4) 3で絞り込んだジョインポイントに対してアドバイスを作用させる。
(5) ウィーブ（アスペクトの合成）後のXML内部モデルをpromelaへ変換する。

4.2 内部モデル

本言語処理系では、promelaとアスペクトを直接ウィーブするのではなく、promelaをXMLの内部モデルに変換し、XMLモデル上でアスペクトを作用させている。

言語処理系の内部モデルは、先の文法を実現するために次のような条件を満たしている必要があった。

(1) 提案言語の文法は、promelaの言語要素に注目したものであり、promelaから内部モデルに変換した際に、そのpromelaの各コードに対して、そのコードがどの言語要素なのかを識別するための属性を与えられること。
(2) ジョインポイントの集合を扱うために、内部モデルは集合、包含関係を扱えること。

XMLはこの2つの条件を満たしている。また、これに加え内部モデルをXMLにすることにより、次のような利点がある。

(1) XPath式を用いて、ジョインポイントを絞り込むことが出来る。
(2) XMLは標準化されている規格であり、XSLTというXMLモデルの変換用言語が用意されているため、本研究での内部モデルを他の処理系への入力として利用できる可能性がある。
(3) 実装時にデバッグをする際に、モデルがテキストデータで表現されているため、デバッガなどのツールを使わずに内部モデルの状態を知ることができる。

以上の理由から、処理系の内部モデルとしてXMLを採用した。

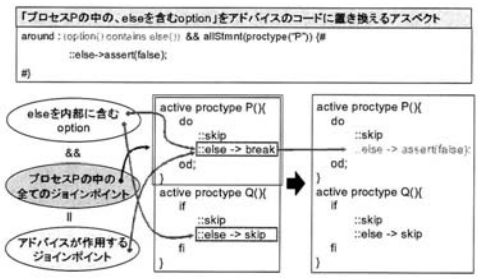

図6 アスペクトの記述例

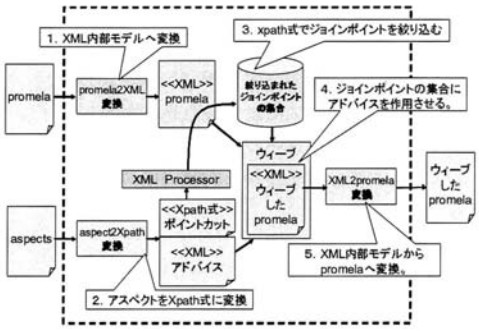

図7 言語処理系の全体図

5. 適用例と考察

適用例として、付録A.1に記載のpromelaモデルに対し3種類の検証を行った。

このpromelaモデルは、プロセスPとプロセスQから構成されQがチャネルに送信したメッセージをPが受信し、そのメッセージにより状態を遷移するというものである。プロセスPの状態は状態1〜3の3状態があり、それぞれをラベルSTATE1〜3で表現し、各ラベルへのgoto文で状態遷移を表現している。各状態では、メッセージ受信用変数rcvMsgにメッセージを受信し、遷移先をif文により判断する。基本的にはプロセスPはmsg1〜3のメッセージを受信することを想定しているが、外部環境によりこれ以外のメッセージを受信することがあるためその場合elseにより状態1に遷移するようにモデル化されている。

このモデルに対して次の3種類の検証を行う。

(1) チャネル受信後の状態の確認
(2) elseを含むoptionの置き換えによる、考慮漏れの確認
(3) goto文の置き換えによる、モデル中の状態1の変更

(1)〜(3)の検証のためのアドバイスはそれぞれ、図8のアドバイス1〜3である。

5.1 検 証 1

検証1は、チャネル受信後のチャネル受信用変数rcvMsgの状態をアサーションにより確認するものである。アドバイスではchanポイントカットを用いて

チャネル ch からの受信操作文*1のジョインポイントに対して、その事後に assert 文を割り込ませるため、チャネル受信後の状態を assert 文により確認することができる。

5.2 検 証 2

検証 2 は、モデル中の全ての else を含むオプションをアドバイスに置き換えるというものである。各状態において、考慮されていないメッセージを受信した場合に else のガードが選ばれるためこの else のガードが選ばれた際に assert 文によりこの状態を検出することができる。

5.3 検 証 3

このモデルは、状態 2、3 では各状態において msg2,msg3 を受信した場合には自状態以外にランダムに遷移するようにモデル化されているが、状態 1 においては msg1 を受信すると自己遷移するようにモデル化されている。アドバイス 3 は状態 1 において msg1 を受信した場合に、状態 2，3 と同じように自状態以外にランダムに遷移する様に状態 1 を変更するためのものである。このアドバイスを適用することにより、状態 1 を変更して検証を行うことができる。

図 8 適用したアスペクト

```
//アドバイス1
after : chan("ch","?*","") {#
  assert(rcvMsg==msg1||rcvMsg==msg2||rcvMsg==msg3)
#}


//アドバイス2
around : option() contains else(){#
  ::else -> assert(false)
#}


//アドバイス3
around : (option() contains expr("rcvMsg == msg1"))
                    && allStmnt(label("STATE1")){#
  ::rcvMsg == msg1 ->
    if
      ::true -> goto STATE2
      ::true -> goto STATE3
    fi
#}
```

5.4 結 果

各アドバイスを promela モデルに適用することにより、各検証を行うための promela モデルを作り出すことが出来た。

検証 1〜3 のうち、検証 1 に関しては横断的な promela モデルの変更を行わずに、不変表明を用いることにより検証を行うこともできる。不変表明とは、図 9 のような記述をモデルに追加することにより、モデルの全複合状態で assert 文の論理式が満たされるかを確認するものである。

図 9 不変表明用のプロセス

```
active proctype inv(){
  assert(rcvMsg==msg1||rcvMsg==msg2||rcvMsg==msg3)
}
```

5.5 状態数の比較

不変表明を用いる場合とアスペクトを用いる場合で必要な作業は異なるが、作業量に大きな差は無かった為、それぞれの検証時に必要な状態数を比較することによりアスペクトを用いた手法の有用性を考察した。

表 5 は、不変表明を用いた場合とアスペクトを用いた場合の状態数を比較したものである。「大域」と「プロセス P 内」はそれぞれ、メッセージ受信用変数 rcvMsg が大域変数であった場合とプロセス P 内のローカル変数であった場合を表す。なお、不変表明を用いる場合には不変表明用のプロセスから assert 文で用いる変数を参照する必要があるためメッセージ受信用変数がプロセス P のローカル変数の場合、検証を行うことができないため表中には (不可能) と表記してある。

表 5 状態数の比較

	不変表明	アスペクト
大域	3124	1657
プロセス P 内	(不可能)	753

不変表明を用いる場合には、状態数は約 3000 であった。一方アスペクトを用いる場合にはその約半分である 1500 程度となっている。これは不変表明がモデルの全ての複合状態で assert 文を実行するのに対し、アスペクトを用いた場合にはチャネルの受信事後にのみ assert 文を実行するためである。

また、アスペクトを用いる場合にはチャネル受信用変数が大域変数である必要はない為、この変数を proctype 内のローカル変数として検証を行った。この際の状態数は不変表明を用いる場合の 1/4 程度に抑えら

1 chan ポイントカットは引数に、チャネル名、操作、メッセージをとる。この例での操作のには"?"が記載されているが、これは全ての受信操作（受信、ランダム受信、ポーリング）を表す。

れている。

不変表明はモデルの全複合状態でその論理式が満たされることを保障できる分、「状態数」という面でコストがかかる。今回の場合、アサーションを実行したい場所がチャネル操作の事後と明確だったため、ポイントカットでこれを表現しアスペクトを記述できた。よって、アサーションで検証したい箇所が明確であり、かつその箇所がモデル中に横断している場合、アスペクトを用いる手法は有効である。

6. ま と め

本稿では、検証目的の変化による、モデルの変化に対応する手段として、アスペクト指向技術を用いる方式を提案し、この方式を実現する為のアスペクト指向言語と言語処理系を開発する共に、promela モデルにアスペクトを適用し、その有効性を確認した。

以下は、今後の課題である。

6.1 アドバイスに対する文法上のチェック

提案した言語処理系はアドバイス中の promela コードに対して文法上のチェックを行っていないが、そのチェックを行うように拡張する。そのためには、アスペクトに指定されたポイントカットの種類によって記述することが出来るアドバイスが何かを定義する必要がある。

6.2 具体的な検証手法の提案

本稿では、アスペクト指向言語とその言語処理系を用いることにより検証目的の変化による検証モデルの変化に対応する基本的な考えを提案した。しかし、検証目的がどの用に変化した場合にどのアスペクトを用いれば良いといったような提案言語の有効な使用方法までは言及していない。今後はこの提案言語を用いた具体的な検証手法を検討していきたい。

付 録

A.1 適用例 1 の promela モデル

```
mtype = {msg1,msg2,msg3,msg4,msg5}
chan ch = [2] of {mtype};
mtype rcvMsg;
active  proctype P()
{
STATE1:
  {
    printf("STATE1\n");
    ch?rcvMsg;
    if
      ::rcvMsg == msg1 -> goto STATE1
      ::rcvMsg == msg2 -> goto STATE2
      ::rcvMsg == msg3 -> goto STATE3
      ::else->goto STATE1
    fi
  };
STATE2:
  {
    printf("STATE2\n");
    ch?rcvMsg;
    if
      ::rcvMsg == msg1 -> goto STATE1
      ::rcvMsg == msg2 ->
         if
           ::true -> goto STATE1
           ::true -> goto STATE3
         fi
      ::rcvMsg == msg3 -> goto STATE3
      ::else->goto STATE1
    fi
  };
STATE3:
  {
    printf("STATE3\n");
    ch?rcvMsg;
    if
      ::rcvMsg == msg1 -> goto STATE1
      ::rcvMsg == msg2 -> goto STATE2
      ::rcvMsg == msg3 ->
         if
           ::true-> goto STATE1
           ::true-> goto STATE2
         fi
      ::else->goto STATE1
    fi
  }
}

active proctype Q()
{
  do
    ::ch!msg1
    ::ch!msg2
    ::ch!msg3
    ::ch!msg4
  od

}
```

参 考 文 献

1) Gerard J.Holzmann, The spin model checker: Primer and reference manual, Addison-Wesley Pub , September , 2003
2) E. M. Clarke, Orna Grumberg, Doron Peled, "Model Checking", MIT Press 2000, 1, 7
3) Gerard J.Holzmann, "コンピュータプロトコルの設計法", 株式会社カットシステム 1994, 11
4) The Eclipse Foundation: "The AspectJ Project", (http://www.eclipse.org/aspectj/)