ソースコードからの決定表生成における冗長性除去技術

小山恭平† 安田和矢† 佐藤直人† 伊藤信治†
篠原宗司† 平田久也† 松尾努†

概要：仕様変更などに伴うシステム改修においては，影響調査を行うための現状仕様の把握が必要不可欠である．しかし，長期運用された大規模システムでは，現状仕様を把握している有識者の不在，設計書の記載不備，ソースコードの複雑化などにより，現状仕様の把握に膨大な時間が必要となっている．仕様把握を効率化する技術のひとつとして，ソースコードからルールを抽出し，決定表化する技術が存在する．しかし，大規模なソースコードでは大量のルールが抽出され，決定表が巨大化するという課題がある．本稿では，抽出したルールに冗長な要素があることに着眼し，それらを除去することで小規模な決定表を生成する技術を提案する．これにより，現状仕様の把握を効率化する．

キーワード：決定表，冗長性，充足可能性問題，SMT ソルバ

Redundancy Elimination for Decision Tables Made with Source Codes

KYOHEI OYAMA† KAZUYA YASUDA† NAOTO SATO† SHINJI ITO†
MOTOSHI SHINOHARA† HISAYA HIRATA† TSUTOMU MATSUO†

Abstract: An objective of this paper is to make analysis of legacy system's specifications efficient. The analysis is one of the most important activity in maintenance. In maintenance of legacy systems, it is one of the hardest activity because of no experts, incorrect documents, less readability of source codes and so on. A conventional technology extracts rules from source codes and visualizes them as decision tables. The technology makes the analysis efficient. However, It may make large-scale decision tables from large-scale source codes. We found that extracted rules have many redundancy items. Our technology eliminates them from the rules. This elimination technology can make smaller decision tables than the conventional technology does and makes the analysis more efficient than it does.

Key Words：Decision Table, Redundancy， SAT Problem， SMT Solver

1. はじめに

現在のソフトウェア開発は差分・派生開発や保守開発などの既存システム改修が全体の約40%を占めるといわれている[1]．既存システム改修においては，影響調査を行うための現状仕様把握が必要不可欠である．しかし，長期間運用された大規模システムでは，現状仕様を把握している有識者の不在，設計書の記載不備，ソースコードの複雑化などにより，現状仕様の把握に膨大な時間が必要となっている．

仕様把握を効率化する技術として，ソースコードからルールを抽出し，決定表化する従来技術が存在する[2]．従来技術では，記号実行[3]を用いてプログラムの実行をシミュレーションすることで，パスの実行条件と対応する変数(以下，出力変数)への値割り当てをルールとして，ソースコードから網羅的に抽出する．抽出したルールを表形式で可視化することで，ソースコードを決定表化する．しかし，条件分岐が多い場合，巨大な決定表が生成されるという課題が存在する．

小規模な決定表を生成する手段のひとつとして，プログラムスライシング[4]を用いて，出力変数の値に影響を与えない命令をソースコードから除去する方法が存在する[5]．しかし，プログラムスライシングを適用しても，巨大な決定表が生成されるケースが存在する．巨大な決定表が生成された場合，それを人手で確認するのに多くの時間が必要となる．

本稿では，抽出したルールに冗長な要素があることに着眼し，それらを除去することで小規模な決定表を生成する技術を提案する．これにより，現状仕様の把握をより効率化する．

2. ソースコードからの決定表生成

2.1 決定表とは

Object Management Group(OMG)によると，決定表とは，入力と出力の関係を表形式で表現したものである[6]．決定表は，複雑な条件判定を含む処理を明確に表現する手段として広く利用されている[7]．決定表の標準記法として，Decision Model and Notation[6]が存在する．

ⓒ2018 Information Processing Society of Japan

図 1は決定表の例である．この決定表は，入力変数 x，y，z の値と出力変数 out の値の対応関係を表しており，各行が入力条件と出力結果を表す論理式からなるひとつのルールに対応している．たとえば，1 行目は「If x < 0 and y > 0 and x > y and z = True Then out = 1」というルールである．各行は入力条件と出力結果から構成されており，各セルには，比較式の真偽値，入力変数の値や値域，出力変数の値や計算式などが記入される．このとき，セル中の要素は，ルールに含まれる比較式，もしくはその否定によって決まる．たとえば，1 行目の入力変数 x と対応する値域「<0」は比較式「x<0」から，比較式「x>y」に対応する真偽「True」は比較式「x>y」から，導出されている．以降では，比較式もしくはその否定のことをリテラルと呼ぶこととする．また「-」は，対応する入力変数の値や比較式の真偽に関係なく，出力変数の値が決まることを意味している．

入力変数名または比較式　出力変数名　ルール

	入力条件				出力結果
	x	y	x>y	z	out
1	<0	>0	True	True	1
2	<0	>0	True	!=True	2
3	<0	<=0	-	True	5
4	<0	<=0	-	!=True	6
7	>=0	-	-	True	0
8	>=0	-	-	!=True	0

ルール番号　入力値域または真偽　出力値または計算式

図 1 決定表の例

2.2 決定表生成の流れ

ソースコードからの決定表生成は，従来技術を活用することで次の 5 ステップにより実現できる．

Step1. スライシング
Step2. ルール抽出
Step3. 条件の論理和展開
Step4. 充足不能なルールの除去
Step5. 決定表化

以降では，図 2 に示す出力変数 out に関するサンプルコードを使って，各ステップの概要を説明する．

```
1  int Sample(int x, int y, bool z){
2    int out = 0, a = 0, b = 0, c = 0;
3    if (x < 0){
4      if(y > 0){
5        if(x < y){
6          a = 1; b = 2;
7        }else{
8          a = 3; b = 4;
9        }
10     }else{
11       a = 5; b = 6;
12     }
13   }else{
14     if(y > 100){
15       c = 0;
16     }else{
17       c = 1;
18     }
19   }
20   if (z == True){
21     out = a;
22   }else{
23     out = b;
24   }
25   return out;
26 }
```

図 2 サンプルコード

2.2.1 スライシング

決定表生成では，まず，ソースコードにプログラムスライシングを適用して，不要な命令を削除する．プログラムスライシングとは，プログラム内の任意の文の変数に影響を与えるコードを元々のプログラムから抽出する技術である[4]．

図 3 はサンプルコードに対してプログラムスライシングを適用し，最終行(25 行目)の出力変数 out に影響を与える命令だけを抽出した結果である．サンプルコードから，変数 c の値を決定する命令(14 行目～18 行目)は，明らかに出力変数 out の値に影響を与えない命令である．そのため，サンプルコードプログラムスライシングを適用することで，14 行目～18 行目が削除されたソースコードが生成される．

ⓒ2018 Information Processing Society of Japan

```
1  int Sample(int x, int y, bool z){
2      int out = 0, a = 0, b = 0, c = 0;
3      if (x < 0){
4          if(y > 0){
5              if(x < y){
6                  a = 1; b = 2;
7              }else{
8                  a = 3; b = 4;
9              }
10         }else{
11             a = 5; b = 6;
12         }
13     }else{
14         if(y > 100){
15             c = 0;
16         }else{
17             c = 1;
18         }
19     }
20     if (z == True){
21         out = a;
22     }else{
23         out = b;
24     }
25     return out;
26 }
```

図 3 スライシング後のサンプルコード

2.2.2 ルール抽出

ルール抽出では，記号実行[3]を用いてプログラムの実行をシミュレートし，パスの実行条件と出力変数 out への値割り当ての組をルールとして網羅的に抽出する．

図 4 はスライシング後のサンプルコードに対して，ルール抽出を適用した結果である．図に示す通り，ルール抽出の結果，8 つのルールがサンプルコードから抽出される．

以降，抽出したルールの集合をルールセットと呼ぶこととする。

1	If (x < 0) and (y > 0) and (x < y) and (z == True)	Then	out = 1
2	If (x < 0) and (y > 0) and (x < y) and not (z == True)	Then	out = 2
3	If (x < 0) and (y > 0) and not (x < y) and (z == True)	Then	out = 3
4	If (x < 0) and (y > 0) and not (x < y) and not (z == True)	Then	out = 4
5	If (x < 0) and not (y > 0) and (z == True)	Then	out = 5
6	If (x < 0) and not (y > 0) and not (z == True)	Then	out = 6
7	If not (x < 0) and (z == True)	Then	out = 0
8	If not (x < 0) and not (z == True)	Then	out = 0

図 4　記号実行によるルール抽出結果

2.2.3 条件の論理和展開

条件の論理和展開では，ルールの入力条件に論理和が含まれる場合，ド・モルガンの法則や分配法則等などを用いて，論理和が含まれない形に変換する．

2.2.4 充足不能なルールの除去

充足不能なルールの除去では，ルールセットに含まれるルールの入力条件が充足可能かどうかを判定し，充足可能性でないルールを除去する．

図 5 は充足不能なルールを除外した結果である．抽出されたルールセットにおいて，ルール 3 とルール 4 の入力条件の一部に着目すると，「(x<0) and (y>0) and not(x<y)」となっており，この論理式は，充足不能なので，ルール 3 とルール 4 を除去する．

1	If (x < 0) and (y > 0) and (x < y) and (z == True)	Then	out = 1
2	If (x < 0) and (y > 0) and (x < y) and not (z == True)	Then	out = 2
5	If (x < 0) and not (y > 0) and (z == True)	Then	out = 5
6	If (x < 0) and not (y > 0) and not (z == True)	Then	out = 6
7	If not (x < 0) and (z == True)	Then	out = 0
8	If not (x < 0) and not (z == True)	Then	out = 0

図 5 充足不能なルール除去後のルールセット

2.2.5 決定表化

決定表化では，ルールセットを決定表の形式に変換する．

図 6 は，図 5 のルールセットを決定表の形式に変換した結果である．このとき，ひとつのルールが決定表の 1 行に対応する．また，ルールが持つリテラルから，決定表のセルに記入する値域や真偽値を決定する．

	入力条件				出力結果
	x	y	x<y	z	out
1	<0	>0	True	True	1
2	<0	>0	True	!=True	2
3	<0	<=0	-	True	5
4	<0	<=0	-	!=True	6
7	>=0	-	-	True	0
8	>=0	-	-	!=True	0

図 6 サンプルコードから生成した決定表

2.3 決定表生成の課題

前節で述べた従来手法では，大規模な決定表が生成され，確認に多くの時間を要する場合がある．

決定表のセルには，値，値域，真偽，計算式，もしくは「-」が記入される．このとき，「-」はそのセルを読み飛ばすことを意味するので，決定表の確認に掛かる時間は，「-」ではないセルの数に依存する．このセル数は，ルールセットに含まれるリテラル総数に依存する．「-」でないセルに記入される内容は，ルールセットに含まれるルールが持つリテラルによって決まる．つまり，ルールセットのリテラル総数が多いと，「-」ではないセルの数が多くなり，確認に多くの時間を要する大規模な決定表が生成される．

従来手法でルール抽出をすると，リテラル総数が多いルールセットが抽出される．リテラル総数は，抽出されたルールセットのルール数とルールに含まれるリテラル数で決まる．このとき，ルールセットのルール数は，条件分岐の数に対して，指数関数的に増える．記号実行を適用すると，単純には，スライシングで残った条件分岐の組み合わせ分だけ，ルールが抽出される．また，ひとつのルールのリテラル数は，条件分岐の数や条件分岐が含むリテラルに依存して増える．ひとつのルールは入力条件と出力結果から構

成される．出力結果は，「出力変数＝変数値」もしくは「出力変数＝計算式」形式の単一のリテラルだけを持つため，リテラル数は固定である．一方，入力条件は，実行パスに存在する条件分岐から構成されるため，条件分岐の数や条件分岐が含むリテラル数に依存して，リテラル数が増える．

たとえば，ある公共システムのソースコードでは，約 100 行のコードから約 4,500 個のルールを含むルールセットが抽出される．このとき，リテラル総数は約 100,000 個である．つまり，このルールセットを決定表化すると単純には，行数 4,500，「-」を除いたセル数 100,000 の決定表が生成されることになる．この決定表を人手で確認するには，多くの時間を要する．

提案手法では，抽出されたルールセットに含まれるリテラル総数を減らすことで，規模が小さい決定表を生成する．

3. 冗長性除去技術

3.1 アプローチ

従来手法で抽出されたルールセットには，出力変数の値に影響を与えない冗長なリテラルが存在することに着眼し，これらを除去することで，リテラル総数を減らし，規模が小さい決定表を生成する．

本稿で対象とする冗長なリテラルは，以下に示すふたつである．

- 自明なリテラル
- ドントケア

自明なリテラルとは，他のリテラルが真ならば，自身も必ず真となるリテラルのことである．図 7 は，自明なリテラル「x < y」の例である．このサンプルコードの 3 行目と 4 行目に着目すると，3 行目の If 文の「x<0 and y>0」が真の場合，4 行目の If 文の 「x<y」は必ず真となる．

```
1  int Sample(int x,  int y){
2     int out = 0;
3     if ((x < 0) and (y > 0)){
4        if(x < y){
5           out = 1
6        }else{
7           out = 2
8     }
9     return out;
10 }
```

Step1.~Step4を適用

1	If (x<0)and(y>0)and(**x < y**)	Then	out=1
2	If not (x < 0)	Then	out=0
3	If not (y > 0)	Then	out=0

図 7 自明なリテラル「x<y」の例

ドントケアとは，そのリテラルの真偽に関わらず，出力変数の値が変わらないリテラルのことである．図 8 はドントケア「y==True」の例である．このサンプルケースの 6 行目から 10 行目に着目すると，「x >0」でないとき，変数 a と変数 b の値は 0 なので，6 行目の「y ==True」が真なのかどうかに関わらず，変数 out の値は 0 である．

```
1  int Sample(int x, bool y){
2     int out = 0, a = 0, b = 0;
3     if (x > 0){
4        a = 1; b = 2;
5     }
6     if (y == True){
7        out = a;
8     }else{
9        out = b;
10    }
11    return out;
12 }
```

Step1.~Step4を適用

1	If (x>0) and (y==True)	Then	out=1
2	If (x>0) and not (y==True)	Then	out=2
3	If not (x>0) and **(y==True)**	Then	out=0
4	If not (x>0) and **not (y==True)**	Then	out=0

図 8 ドントケア「y ==True」の例

本稿では，前述したふたつの冗長なリテラル（自明なリテラル，ドントケア）を除去するため，次の 4 ステップにより決定表を生成する．

Step1. スライシング

Step2. ルール抽出

Step3. 冗長なリテラルの除去

Step4. 決定表化

従来手法の「条件の論理和展開」と「充足不能なルールの除去」に変わり，「冗長なリテラル除去」を行う．冗長なリテラル除去は，処理の過程で充足不能なルールの除去と条件の論理和展開ができる．以降では，提案手法の 4 ステップのうち，冗長なリテラル除去の手法について説明する．

冗長なリテラル除去では，出力結果が同じルールの入力条件の集合 IN に包含され，かつ，冗長なリテラルを含まない入力条件を生成する．このような入力条件を生成するため，式(1)が恒真となる入力リテラル集合 LS から，除去しても式(1)が恒真となるリテラルをすべて除去することで，極小なリテラル集合を導出する．極小なリテラル集合のリテラルを論理積で結合することで，新たな入力条件を生成する．

$$\left(\bigwedge_{l \in LS} l\right) \rightarrow \left(\bigvee_{in \in IN} in\right) \qquad (1)$$

極小なリテラル集合とは，任意のリテラルを除去すると，除去後のリテラル集合と IN において，式(1)が恒真とならないリテラルの集合のことである．

ここで，LS から除去しても式(1)が恒真となるリテラル l は冗長なリテラルである．IN には出力結果が同じ入力条件がすべて含まれている．このとき，リテラル集合 LS と IN において，式(1)が恒真ならば，LS から生成された入力条件 in が真となるとき，IN に含まれるルールのうち，いずれかが真となるので，in は出力結果が同じ入力条件である．ここで，LS からリテラル l を除去した LS’においても式(1)が恒真ならば，LS’から生成した入力条件 in’についても，in’が真となるとき，IN に含まれるルールのうち，いずれかが

真となるので, 出力結果が同じ入力条件である. このとき, リテラル l を除去しても, in と in'で出力結果は同じである. よって, 出力結果が同じなので, リテラル l は出力変数の値に影響を与えない冗長なリテラルである.

3.2 処理概要

図 9 は冗長なリテラル除去のフローチャートである. 本手法では, まず, ルールセットにおいて, 出力結果が同じルールの入力条件をグルーピングし, 入力条件の集合 IN を生成する. 次に, 入力条件の集合 IN ごとに, 新たな入力条件を生成する. 入力条件の生成では, まず, IN から入力条件に使用するリテラルを生成する. 次に, そして, セレクションにより, 新たな入力条件の候補として, 式(1)が恒真となるリテラルの組み合わせを選択する. このとき, そのようなリテラルの組み合わせをすべて生成済みの場合, IN に対する入力条件の生成を終了する. 組み合わせがある場合は, その組み合わせを極小化し, 新たな入力条件の集合 IN'にリテラル集合のリテラルを論理積で結合した論理式を入力条件として追加する. すべての条件の集合 IN に対して, 新たな入力条件 IN'生成したら, ルールセット中の入力条件を新たに生成した入力条件に入れ替える.

以降では, 図 9 の下線が引かれたステップの詳細を説明する.

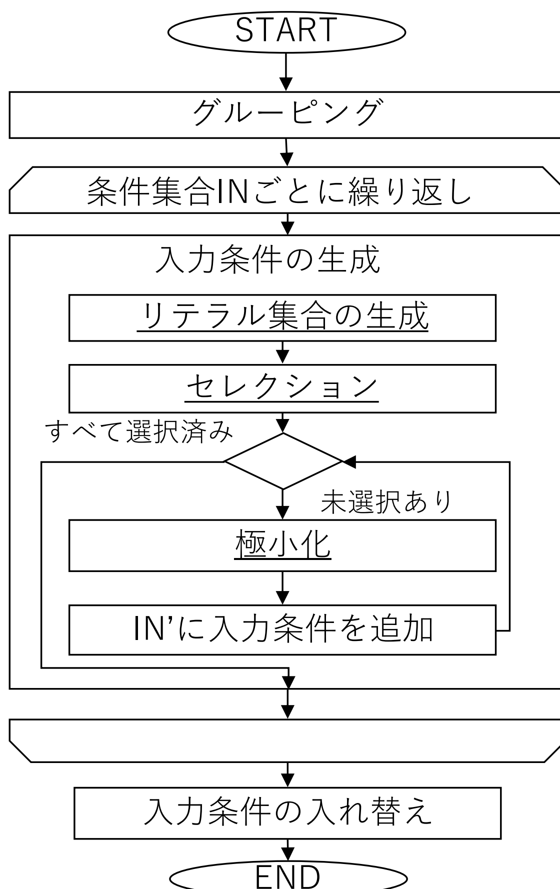

図 9 冗長性除去のフローチャート

3.2.1 リテラル集合の生成

本手法では, IN に含まれるリテラルを組み合わせることで, 新たな入力条件を生成する. そのため, リテラル集合の生成では IN から次の 2 種類のリテラルを生成し, リテラル集合とする.

- IN の条件に含まれる比較式
- IN の条件に含まれる比較式の否定

3.2.2 セレクション

セレクションでは、生成したリテラル集合から新しいルールの入力条件の構成要素となるリテラルの集合を選択する.

まず, 式(2)が充足可能かどうかを判定する. 式(2)は, IN の入力条件のいずれかが真となり, かつ, IN'の入力条件はいずれも真とならないことを意味している. このとき, 式(2)が充足可能ならば, 式(1)が恒真となるリテラル集合 LS のうち, 未選択の集合が存在することを意味する.

$$\left(\bigvee_{in\in IN} in\right)\wedge\neg\left(\bigvee_{in'\in IN'} in'\right) \tag{2}$$

式(2)が充足可能ならば, 真となる値割り当てにおいて, 真となっているリテラルの集合(充足リテラル集合)を取得する. IN'の入力条件は, IN の入力条件に含まれるリテラルから構成され, リテラル集合は IN の入力に含まれる比較式, もしくはその否定を網羅しているので, 充足リテラル集合は, 式(2)が真となる比較式への真偽値の割り当てに対応する. つまり, 充足リテラル集合に含まれるリテラルがすべて真となるとき, 式(2)も真となるので, 充足リテラル集合は, 式(1)が恒真となるリテラル集合 LS のうち, 未選択の集合である.

一方, 充足不能ならば, 式(1)が恒真となるリテラル集合 LS は, すべて選択済みなので, 処理を終了する.

3.2.3 極小化

極小化では 充足リテラル集合を極小なリテラル集合にするため, 任意の真部分集合において, 式(1)が恒真とならないように, リテラルを除去する. 本手法では任意の順番で充足リテラル集合から, ひとつのリテラル l を除外して, 充足リテラル集合の真部分集合 LS'を作る. 次に式(3)を充足可能性問題として解く. 式(3)は, 式(1)の LS を LS'に変え, 全体を否定した式であり, LS'に含まれるリテラルがすべて真となるとき, IN の入力条件がすべて真にならないことを意味している. この式が充足可能な場合, LS'は式(1)が恒真となるリテラル集合 LS ではない. 任意の LS'において, 式(3)が充足可能な場合, 充足リテラル集合は, 式(1)が恒真となる極小なリテラル集合である.

$$\left(\bigwedge_{l\in LS'} l\right)\wedge\neg\left(\bigvee_{in\in IN} in\right) \tag{3}$$

式(3)が充足可能あれば, 除去したリテラル l を充足リテラル集合に戻す. 一方, 充足不能であれば, LS'が IN の入力条件に包含されてしまうので, 充足リテラル集合に戻さ

ⓒ2018 Information Processing Society of Japan

ない．このチェックを充足リテラル集合のすべてのリテラルに適用することで，充足リテラル集合を極小なリテラル集合にする．

4. 評価

本章では，公共系のシステムのソースコードに対して，冗長性除去技術を用いた決定表生成を適用した結果について述べる．

4.1 題材

本評価では，表 1に示す10個のソースコードに対して，従来手法の Step1～Step4 を適用して抽出したルールセットと提案手法の Step1～Step3 を適用して抽出したルールセットのルール数及びリテラル総数を比較する．ここで，表 1の各ソースコードは，公共系システムで実際に使用しているものである．また，表 1 の LOC は，ソースコードに，スライシングを適用し，出力変数の値に関係のない命令を除去した後の行数である．

表 1 評価題材の一覧

No	ファイル名	LOC
1	P1	3
2	P2	8
3	P3	9
4	P4	22
5	P5	47
6	P6	78
7	P7	83
8	P8	88
9	P9	99
10	P10	102

4.2 結果

評価の結果，提案手法により，ルールセットのリテラル総数及びルール数が少なくできることを確認した．

表 2は，従来手法と提案手法を用いて抽出したルールセットの規模の比較である．この表において，下線を引いている数値は，従来手法で抽出したルールセットよりも提案手法で抽出したルールセットの方が，値が小さいことを示している．

表 2から，10 ケース中 6 ケースで，提案手法により，リテラル総数及びルール数が少なくなっている．リテラル総数及びルール数が減っているので，適用後のルールセットを決定表化することで，「"-"」でないセルの数や行数が少ない決定表を生成可能である．

また，10 ケース中 3 ケースで，従来手法と提案手法で抽出されたルールセットのルール数とリテラル総数が同じになっている．このケースでは，ソースコードから抽出したルールセットに冗長なリテラルが存在しておらず，提案手法でも従来手法と同じルールセットが抽出されている．

さらに，10 ケース中 1 ケースで提案手法の方が，リテラル総数及びルール数が多くなっている．

表 2 ルールセットの規模比較

No	従来手法		提案手法	
	ルール数	リテラル総数	ルール数	リテラル総数
1	2	2	2	2
2	1	1	1	1
3	13	32	<u>7</u>	<u>18</u>
4	12	31	12	31
5	18	72	<u>16</u>	<u>65</u>
6	506	6,880	<u>222</u>	<u>1,149</u>
7	794	12,080	<u>396</u>	<u>3,722</u>
8	48	212	1,373	5,457
9	362	4,512	<u>190</u>	<u>1,004</u>
10	4,506	106,240	<u>673</u>	<u>8,897</u>

5. おわりに

5.1 結論

本稿では，ソースコードからの決定表生成における決定表の巨大化という課題の解決を目的に，抽出されたルールセットに含まれる入力条件から冗長なリテラルを除去する技術を提案した．提案手法では，出力結果が同じルールの入力条件の集合に対して，それらに包含される極小なリテラル集合を導出することで，冗長なリテラルを含まない新たな入力条件を生成する．

公共システムの実ソースコードに対して，従来手法と提案手法を用いてルールセットを抽出し，その規模を比較した．その結果 10 ケース中 6 ケースで，提案手法を用いて抽出したルールセットの方が，総リテラル数及び行数が少ないことを確認した．決定表の規模はルールセットの総リテラル数に依存することから，提案手法により決定表の規模縮小が見込める．

5.2 今後の課題

提案手法の方が従来手法よりもルールセットの規模が大きくなるケースが発生している．今後，この原因を分析し，対策を行う．

参考文献

[1]情報処理推進機構：2012 年度「ソフトウェア産業の実態把握に関する調査」調査報告書，https://www.ipa.go.jp/files/000026799.pdf，2013/04(参照：2018/01/05)
[2]酒井政裕，岩政幹人：記号実行によるプログラム改造支援技術，東芝レビュー Vol.67 No.12，2012/12
[3]King, J. C.：Symbolic execution and program testing, Communications of the ACM．19, 7, pp385-394, 1976
[4]Weiser, Mark：Program Slicing, IEEE Transaction on Software Engineering, Vol10, No.4, pp.352-357, 1984
[5]Tomomi Hatano，Takashi Ishio，Joji Okada，Yuji Sakata，Katsuro

Inoue：Extraction of Conditional Statements for Understanding Business Rules，2014 6th International Workshop on Empirical Software Engineering in Practice 2014
[6]Object Management Group：Decision Model and Notation(DMN) v1.1，http://www.omg.org/spec/DMN/1.1/ ，2016/06，(参照：2018/01/05)
[7]斉藤剛：決定表の演算とその応用，情報処理会論文誌，27(3)，p281-288，1986/03

ⓒ2018 Information Processing Society of Japan