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

2004－ＳＬＤＭ－115　（８）
2004／5／28

抽象解釈手法に基づく変数の相互関係解析とそのデータパス最適化への応用

土井　伸洋†　　堀山　貴史††　　中西　正樹†††　　木村　晋二†

† 早稲田大学大学院 情報生産システム研究科　〒808-0135 北九州市若松区ひびきの 2-7
†† 京都大学大学院 情報学研究科　〒606-8501 京都市左京区吉田本町
††† 奈良先端科学技術大学院大学 情報科学研究科　〒630-0101 奈良県生駒市高山町 8916-5
E-mail: †nobuhiro_doi@fuji.waseda.jp, ††horiyama@i.kyoto-u.ac.jp, †††m-naka@is.naist.jp,
††††shinji_kimura@waseda.jp

あらまし　Cプログラムからのハードウェア合成においてはビット長最適化をはじめとするさまざまなハードウェア向け最適化が必要である．このためにはプログラム中の変数がとりうる値やデータフローを推測することが必要で，静的解析手法が使われることが多いが，精度などの点で不十分な点がある．本稿ではソフトウエア検証の分野で注目されている抽象解釈 (Abstract Interpretation) 手法に基づくプログラムの解析と，データパス最適化への応用について述べる．

キーワード　HDL, 高位合成, 並列化コンパイラ, ビット長最適化, 抽象解釈

Program Analysis Based on Abstruct Interpretation and Its Application for Datapath Optimization

Nobuhiro DOI†, Takashi HORIYAMA††, Masaki NAKANISHI†††, and Shinji KIMURA†

† Graduate School of Information, Production and Systems, Waseda University
Hibikino 2-7, Wakamatsu-ku, Kitakyushu 808-0135 Japan
†† Graduate School of Informatics, Kyoto University
Yoshida-honmachi, Sakyo-ku, Kyoto, 606-8501 Japan
††† Graduate School of Information Science Nara Institute of Science and Technology
8916-5 Takayama-cho, Ikoma, Nara, 630-0101 Japan
E-mail: †nobuhiro_doi@fuji.waseda.jp, ††horiyama@i.kyoto-u.ac.jp, †††m-naka@is.naist.jp,
††††shinji_kimura@waseda.jp

Abstract　Various optimization techniques such as bit-length optimization are required for hardware generation from C programs. The value range analysis and dataflow analysis are effective for such optimization and static program analysis methods have been used. The static methods, however, have several problems such as the preciseness, the overestimation, etc. In this paper, we describe a program analysis method based on abstract interpretation and its application for datapath optimization.

Key words　HDL, high-level synthesis, parallelizing compiler, bit-length optimization, abstruct interpretation

1. はじめに

今日，LSI は社会のありとあらゆるものに使用されており，LSI がこれらの機器をささえているといっても過言ではない．そして，回路微細化技術の進歩を背景に ますます複雑で高機能な LSI が設計されるようになってきた．だがその回路規模は，従来のハードウェア記述言語を用いた方法では設計が不可能なレベルに達しつつある．さらに，企業間の激しい製品開発競争やコスト削減といった理由から，LSI 設計の効率化が急務となっている．このような状況から，C 言語やその拡張言語などの高級言語からハードウェアを自動的に合成する高位合成が急速に実用化されつつある [1]～[4].

高級言語からのハードウェア合成においてはコンパイラによるハードウェア向け最適化が欠かせない．なぜなら，C 言語な

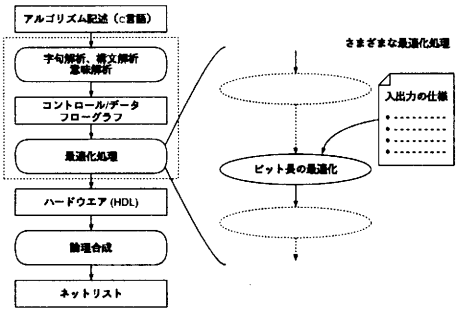

図1　C言語からのハードウエア生成

どの高級言語で書かれたプログラムは逐次処理を前提としており，そのままではハードウェアの利点を生かすことができないためである．効率のよいハードウエアを合成するためには並列化やデータパスの最適化が欠かせない．我々は，変数やデータパスのビット幅に着目し，その最適化ついて研究を行なってきた．特に，プログラム中に浮動小数点演算が含まれる場合に，これを自動的に適切なビット幅をもった固定小数点演算へと変換する方法に焦点をあててきた [5], [6].

これらの最適化を行なうにはプログラム解析を行ない，データフローや変数がとりうる値の範囲を推測することが必要である．このために，静的解析を利用する手法が数多く提案されているが，不定回数のループなど扱うのが困難な構造もある．

一方で，ソフトウェア開発の分野でも，ソフトウェアの仕様検証やデバッグを効率化するために，プログラム解析の重要性が高まってきている．プログラム解析手法としては高位合成と同じように静的解析が利用されることが多かったが，近年，抽象解釈 (Abstract Interpretation) に基づく解析手法が注目されている [7]．これはプログラムのデータ領域や動作を抽象化することで問題を有限化し，そこからプログラムのさまざまな性質を求める方法である．

本稿では，この抽象解釈手法を，高位合成におけるデータパスビット長最適化へ応用する方法について検討を行なった．以下，二章ではコンパイラの概要，三章では抽象解釈について，四章では抽象解釈のビット長最適化への応用について，五章では評価とまとめを示す．

2. C言語からのハードウェア生成

C言語からハードウエアを生成するコンパイラは，図1に示すように，C言語で書かれたソースファイルと，ハードウエアの入出力仕様が書かれた情報ファイルを読み込み，レジスタ転送レベルのHDL記述を出力する．入力仕様ではプログラムからではわからない入力データのビット幅や出力データのビット幅を指定する．また，内部データについても指定が可能である．

コンパイラは入力としてfloatやdoubleといった型を使って記述されたC言語を受けとり，字句や構文の解析の後，プログラムの処理内容を表現するコントロール/データフローグラフを生成する．次に，得られたコントロール/データフローグラフを元にハードウエア向けの最適化を行なう．データパスのビット長最適化処理はこの一部であり，入出力仕様 (例えば出力の精度) を満たすのに必要なレジスタや演算器のビット長を自動的に計算する．最終的に得られたコントロール/データフローグラフと変数や演算器のビット長に関する情報をもとにHDL記述を生成する．本稿では，ビット長最適化処理に必要なコントロール/データフローグラフの解析に抽象解釈を適用する方法について述べる．

3. 抽象解釈

プログラム解析を行なうことで，効率化や並列化を行なったり，プログラム中から意味的な誤りを検出することができる．だが一般的に扱うデータ領域があまりにも広いので，計算機で扱えないことが多い．そこで，抽象解釈手法においては，注目した性質のみを保存し，その他の情報は抽象化してしまうようにする．これによりプログラムを比較的小さな有限領域に持ち込み，その上でプログラムの性質を明らかにする．主に関数型言語をベースとした研究が盛んに行なわれており，その実装としてHaskell [8] などが知られている．

手続き型言語対する実用的な例は関数型言語に比べると少ないが，アクション履歴抽象化やアクション抽象化といったものが提案されている．それぞれについて簡単に述べる．

3.1 アクション履歴抽象化

手続き型プログラムの最適化を行なう時には，プログラムをデータフローグラフなどの非決定フローグラフに変換することが多い．ここでの抽象化では，条件分岐は単なる非決定的な遷移動作に変換される．また，フローグラフの各ノードにおかれていたアクションをそのノードにつながるエッジ上に移す．

アクション履歴抽象化においては，ある経路がどのようなアクションの履歴を持ちうるかをフローグラフをもとに調べる．そして，データフローグラフを有限オートマトンとみなすことによって，正規表現で記述された仕様をみたしているかどうかを有限オートマトンの包含関係に帰着させ判定することができる．

例として，プログラム中で利用するファイルが正しく扱われているかを検証する問題を考える．ファイルの正しく扱うために必要なアクションの集合は

{open, write, close}

であり，その履歴は

(open, write*, close)*

という仕様を満たす必要がある．この仕様は有限オートマトンとして記述することができる (図2)．ここで，状態0は初期状態であるとともに受理状態であり，状態2は変則性を検出した状態である．

ここで，データフローグラフから有限オートマトンを作成する時には {open, write, close} 以外のアクションを，空アクションとして処理することができる．

他にも変数の多重定義，未定義変数の参照などのチェックもアクション履歴抽象化を用いてチェックすることができる．

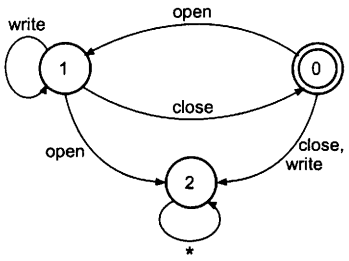

図 2　有限オートマトンで記述された仕様

3.2　アクション抽象化

アクション抽象化は手続き型言語の広域解析法として古くから知られている抽象化手法である．ここでは，値そのものの性質ではなく，値に加えられる操作に着目する．たとえば，代入文"x := a + (b * c)"を，

{(x:def) (a:ref) (b:ref) (c:ref) (b*c:gen) (a+b*c:gen)}

などと抽象化することができる．ここで，:def, :ref, :gen はそれぞれ，値の定義，値の参照，計算結果の生成などを表している．

山岡らは[9] のなかで，アクション抽象化と時相論理を組み合わせ，Java と相互変換可能な Jimple 言語に対する仕様検証を行なう手法を提案している．時相論理とは命題論理に対し時間に関する様相演算子 $F(future), G(generally), X(nexttime), U(until)$ と経路に関する様相演算子 $A(for\ all\ paths), E(for\ some\ paths)$ を追加することで，時相を表現できるようにしたものである．ハードウェア検証の分野では，CTL(Computational Tree Logic) のモデルチェッキングに BDD(Binary Decision Diagram) や SAT を応用し，非常に高速な実行を可能にしたものが提案されている．

[9] のなかでは，プログラムに対する仕様記述を時相論理 CTL-FV で記述しモデルチェッキングを行なうことで，プログラムの検証を行なっている．たとえば，「プログラム中に無用な命令は定義されていない」という仕様を

AG ! (def(x) & AX !EF use(x))

のように記述している．そして，モデルチェッキングに既存のモデル検査ツール SMV を利用することで，比較的大きいサイズのプログラムに対しても高速に処理ができるとしている．

4.　ビット長最適化への応用

ここでは，抽象解釈手法に基づく解析手法をビット長最適化に適用する方法について述べる．適用においては以下の二つの手法を用いる．

- ある変数のビット長推定においては，その変数に関係しない部分は抽象化して隠してしまう．
- 繰返し回数の不明なループ構造に対しては抽象解釈を適用する

これらの手法の内，1 番めの手法は一般的であるので，2 番めの手法について詳しく述べる．

```
double newton(double baseNum){
  double x      = 0.0;
  double x_next = 100.0;
  double eps    = 0.0001

  while (baseNum - x_next*x_next >   eps ||
         baseNum - x_next*x_next < - eps)  {
    x = x_next;
    x_next = x - (x*x-baseNum) / (2*x);
  }
  return x_next;
}
```

図 3　サンプルプログラム

本手法を説明するにあたり，図 3 のプログラムを例とする．このプログラムはニュートン法を利用して，与えられた数の平方根を計算するプログラムである．プログラムの仕様として"baseNum は 1 から 100 までの整数"，"終了条件は，平方根の誤差が eps 未満であること"，"初期値は 100" があるものとする．

このプログラムの中には浮動小数点演算が含まれているので，最適なハードウェア合成のためにはこれらの演算を適切なビット長の固定小数点演算で実行する必要がある．そのために，演算に必要なビット数を見積もらなければならない．しかし，浮動小数点演算を固定小数点演算化すると演算誤差が発生するため，これを考慮してビット長の最適化を行なう必要がある．だが，繰返し回数のわからないループ構造があるような場合は誤差がどの程度増えるか推測できず，今までは取り扱うことができなかった．

本章では，まず我々の提案している誤差のモデルについて説明する．つぎに，繰返し回数不定のループ構造に関する問題点と，抽象解釈法を適用した過程を示す．

4.1　誤差のモデル

変数に含まれる誤差は，丸め誤差と伝搬誤差とでモデル化される．演算の入力に含まれる誤差が演算結果に与える影響が誤差伝搬，それとは別に演算結果を格納する変数のビット長が制限されることによって生成される誤差が丸め誤差である．丸め誤差は，定数を含めて小数点数を固定小数点数で表すときに発生する．

4.1.1　丸め誤差

丸め誤差は小数部を固定長で打ち切ったことにより発生する誤差であり，ある固定小数点 X の小数部ビット長を L_X と表すと，丸め誤差 $E_R(X)$ は以下の式で表される．

$$E_R(X) \quad \leq \quad 2^{-L_X}$$

4.1.2　伝搬誤差

伝搬誤差は，誤差が演算により伝搬されることを示す．演算をすべて二項演算とし，代入先のデスティネーション X_{dst}，演算

の入力であるソースオペランド X_{src1}, X_{src2}, 演算の種類 $\circ$ で

$$X_{\mathrm{dst}} \leftarrow X_{\mathrm{src1}} \circ X_{\mathrm{src2}}$$

と表す. このとき X_{dst} への誤差伝搬 $E_P(X_{\mathrm{dst}})$ は, ソースオペランドの持つ誤差 ΔX_{src1} および ΔX_{src2} および演算の種類に応じて計算することができる. 例えば加算であれば

$$E_P(X_{\mathrm{dst}}) \quad = \quad \Delta X_{\mathrm{src1}} + \Delta X_{\mathrm{src2}}$$

となる. 減算, 乗算, 除算については図 1 に示す. なお, $X.|\max|$, $X.|\min|$ はそれぞれ変数 X の上界および下界の絶対値をとったものである.

丸め誤差と伝搬誤差の両方を含んだ X_{dst} の誤差 ΔX_{dst} は

$$\begin{aligned}\Delta X_{\mathrm{dst}} &= E_R(X_{\mathrm{dst}}) + E_P(X_{\mathrm{dst}}) \\ &= E_R(X_{\mathrm{dst}}) \\ &+ Propagate(X_{\mathrm{src1}}, \Delta X_{\mathrm{src1}}, X_{\mathrm{src2}}, \Delta X_{\mathrm{src2}}, \circ)\end{aligned}$$

と表せる. *Propagate*() は演算 $\circ$ に応じた伝搬誤差の関数を表す.

4.2 ループに対する解析

プログラム中のループ構造は次の 3 種類に分類される.

（1） ループ回数がわかっている

（2） 最大ループ回数だけはわかっている

（3） ループ回数はまったくわからない

ループ構造が (1) もしくは (2) のときはループを展開し通常のコード列にすることで対応する.

(3) の場合にはループ内部のコードに応じてそれぞれ処理を行なう必要がある. 最も困難なのは, ループを重ねるごとに誤差が増大してゆく場合である. 例えばループ中に

x = x ∘ a (ただし x, a は float/double の変数とする)

なる演算 $\circ$ が存在する場合, a の誤差が 0 である場合を除き, 演算を重ねるごとに x には a と演算 $\circ$ による誤差が蓄積してゆく. これにより x, a それぞれのビット長を推測することができず, ハードウェア合成を行なう際には障害となる. そこで, まずはループ内部のコードにおいて, 変数がどのように定義され参照されているかを抽象化を用いて表現する.

変数に加えられる操作に注目し, アクションを次のように抽象化する.

= ··· 値の定義 (代入)

∘ ··· 通常の演算 (四則演算, シフト)

◇ ··· 比較演算 (比較演算子)

また変数そのものの性質を次のように抽象化する.

int ··· 整数 (char/int が対応)

flt ··· 浮動小数点数 (float/double が対応)

すると, 問題の発生するコード列は図 4 のように表すことができる. 正確には, ループ中に同じ変数への参照と代入が行なわれている点, そしてそれらの他に誤差をもつ変数との演算 $\circ$ が行なわれていることである.

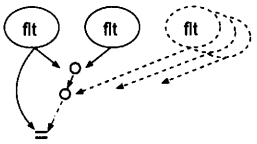

図 4 誤差が増大してゆく場合のループ構造

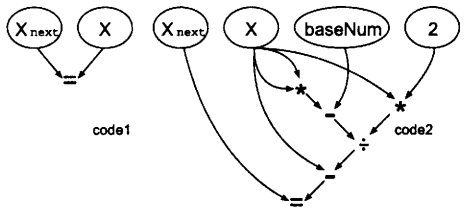

図 5 例題における変数の相互関係

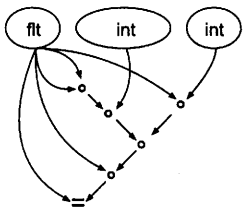

図 6 抽象化された構造

例題におけるループ内部の変数は, 図 5 ような関係にある. 図 5 のなかでは $X_{\mathrm{next}} = X$ の構造があるので, これを code 2 のフローグラフと合成する. さらに, 定義の基づいて演算を抽象化すると図 6 のようになる.

よって例題に関してもループ中に同じ変数への参照と代入が行なわれていると同時に誤差をもつ変数との演算が行なわれており, ビット長最適化の際には問題を引き起こす. 次章では, このような場合に対処する方法について述べる.

4.3 ビット長の推測

ループ中のコードからは誤差が増大する一方であることしか判断できないので, このままではビット長最適化を行なうことは不可能である. そこで, 今度は終了条件に着目する. ループの繰返し回数が不明であっても, ループの終了条件などから必要なビット長を推定することが可能である場合もありうる.

先にも述べたとおり, ビット長最適化が不可能であるのは x = x ∘ a における x および a のビット長が推測不能なためである. だが次のような構造について考えてみる.

```
while (x ◇ b)
    x = x ∘ a
```

ループの終了条件として変数 b との比較演算が定義されている. この値を元に, x のビット長を推測することができれば, 繰返し回数が不定であっても推測が可能となる. このことをフローグ

表 1　伝搬誤差の見積り式

演算	伝搬誤差の大きさ
$X_{\mathrm{src1}} \pm X_{\mathrm{src2}}$	$E_P(X_{\mathrm{dst}}) = \Delta X_{\mathrm{src1}} + \Delta X_{\mathrm{src2}}$
$X_{\mathrm{src1}} \times X_{\mathrm{src2}}$	$E_P(X_{\mathrm{dst}}) = X_{\mathrm{src1}}.\lvert\max\rvert \cdot \Delta X_{\mathrm{src2}} + X_{\mathrm{src2}}.\lvert\max\rvert \cdot \Delta X_{\mathrm{src1}} + \Delta X_{\mathrm{src1}} \cdot \Delta X_{\mathrm{src2}}$
$X_{\mathrm{src1}} \div X_{\mathrm{src2}}$	$E_P(X_{\mathrm{dst}}) = \dfrac{\Delta X_{\mathrm{src1}}}{X_{\mathrm{src2}}.\lvert\min\rvert}$　(除数が 0 の時は $X_{\mathrm{src2}}.\lvert\min\rvert = 2^{-L_{src2}}$ とする)

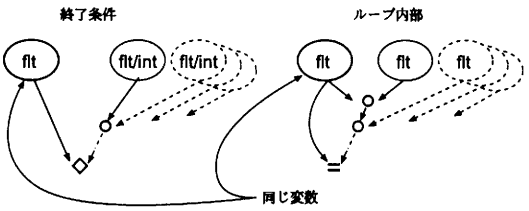

図 7　条件部からビット長推測ができる可能性がある場合

ラフで表現すると図 7 となる.

例題におけるループの終了条件は平方根の誤差が 0.0001 未満になった時であり, そのコードは

```
while (baseNum - x_next * x_next >   eps ||
       baseNum - x_next * x_next < - eps)
```

である. したがって, この判定が誤りなく行なわれるには baseNum - x_next * x_next の演算結果が常に誤差 eps 以下であることが必要となる. 変数 baseNum は整数であるため, 誤差は含まれない. よって, X_{next} の二乗による誤差が eps 未満でなければならなず, 表 1 より

$$0.0001 > 2 * (X_{\mathrm{next}}.\lvert\max\rvert * \Delta X_{\mathrm{next}}) + {\Delta X_{\mathrm{next}}}^2$$

を満たさなければならないことがわかる.

上記の条件からビット長を推測するためには, 変数 X_{next} がとりうる最大値 $X_{\mathrm{next}}.\lvert\max\rvert$ を求めることが必要である. だが, 変数 X_{next} を計算するために変数 X を使用しており, X は X_{next} そのものに他ならない. よって, これらの条件から $X_{\mathrm{next}}.\lvert\max\rvert$ を求めることは不可能である.

ただし, これを入力仕様として与えることで推測することは可能である. ニュートン法の性質上, 変数 X_{next} は初期値をスタートとして与えられた数の平方根へと収束してゆく (図 8). 例題では"baseNum は 1 から 100 までの整数", "初期値は 100" 等の条件があるため変数 $X_{\mathrm{next}}.\lvert\max\rvert = 100$ であることが推測できる. そこで, この式を新に条件として入力する.

$\Delta X_{\mathrm{next}} \gg {\Delta X_{\mathrm{next}}}^2$ より ${\Delta X_{\mathrm{next}}}^2$ の項を無視できると考えると, 条件式は $10^{-6} > 2 * \Delta X_{\mathrm{next}}$ となる. これを満たすのに必要なビット長は 21 bit となる.

この推測が正しいかどうかを確かめるために, シミュレーション用い手作業により必要なビット長を求めてみた. その結果, 変数 X_{next} に必要なビット長は 21 bit であることがわかった. これにより, 提案手法による解析が妥当であることがわかった.

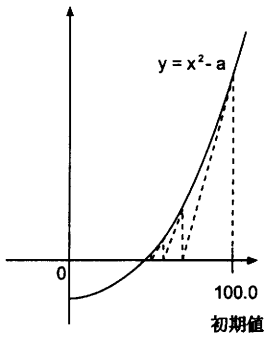

図 8　ニュートン法における収束の様子

5. ま と め

本稿では浮動小数点演算を含んだ C プログラムからハードウェアを合成する際に鍵となる浮動小数点演算の固定小数点演算化について, 繰返し回数不定のループ構造に対する解析を試みた. その結果として, 抽象化手法にもとづいて合成の際に問題となるループ構造を抽出し, 入力仕様で指定された入出力のビット長から, 繰返し回数が不定のループに対してもビット長を推測できることを示した.

今後の課題としてはループ構造とその条件判定部に関する詳細な解析があげられる. 本稿では, 不定回数のループに対するビット長の推定方法を示したが, その終了条件は, ある一定の定数で限定されているという形をしていた. このような構造すべてのループに対して本手法が適用できるかどうかは今後の検討が必要である. また, 本手法を拡張し, より一般的なループに対しても適用できる手法を考える必要がある.

また, 指数部の大きく異なる浮動小数点演算が現れるようなプログラムの解析やソフトウェアを含めたシステム検証に抽象化手法を応用することについても検討してゆく予定である.

謝　辞

本研究を進めるにあたり日頃から有益なご助言, ご指導を頂いた早稲田大学大学情報生産システム研究科の皆様に深く感謝します. また, 本研究は一部日本学術振興会科学研究費補助金, NEC, 文部科学省北九州知的クラスタープロジェクト研究費補助金による.

文　献

[1] Akihisa Yamada, Ryoji Sakurai, Masayuki Yamaguchi, Takashi Kambe, and Hiroyuki Katata. "'Hardware synthesis with the Bach system". In *IEEE ISCAS'99*, pages

366–369, May 1999.

[2] Kazutoshi WAKABAYASHI. "C-Based Synthesis Experiences with a Behavior Synthesizer "Cyber"". In *DATA'99*, pages 390–393, January 1999.

[3] The Open SystemC Initiative. http://www.systemc.org.

[4] SpecC Technology Open Consortium. http://www.SpecC.org.

[5] Osamu Ogawa, Kazuyoshi Takagi, Yasufumi Itoh, Shinji Kimura, and Katsumasa Watanabe. "Hardware Synthesis from C Programs with Estimation of Bit Length of Variables". *IEICE Transaction*, E82-A(11):2338–2346, November 1999.

[6] Nobuhiro DOI, Takashi Horiyama, Masaki Nakanishi, Shinji Kimura, and Katsumasa Watanabe. "Bit Length Optimization of Fractional Part on Floating to Fixed Point Conversion for High-Level Synthesis". *IEICE Transactions on Fundamentals*, E86-A(12):3176–3183, December 2003.

[7] 小川瑞史, 小野論. "抽象実行 そのフレームワークと実例 (その1〜3)". **コンピューターソフトウェア**, 13(2):3–18, 13(4):3–22, 13(6):3–25, 1996.

[8] Charles Consel. "Fast Strictness Analysis via Symbolic Fixpoint Iteration". In *International Static Analysis Symposium (SAS '94)*, pages 423–431, 1994.

[9] 山岡裕司, 胡振江, 武市正人, 小川瑞史. "モデル検査技術を利用したプログラム解析器の生成ツール". **情報処理学会論文誌:プログラミング**, 44(SIG13):25–37, 2003.