抽象化を用いた検証ツールの調査*

田辺 良則　　高井 利憲　　高橋 孝一

科学技術振興機構, CREST　　産業技術総合研究所

{tanabe.yoshinori,t-takai,k.takahashi}@aist.go.jp

概 要

モデル検査技法は、設計の仕様に対する妥当性検証への適用において、近年大きな成功をおさめている。この技法の適用範囲をさらに広げるためには、状態数爆発問題を解決することが必要である。この問題を解決する方法として注目されている抽象化技法、およびそれを実装したツールについて調査を行った結果を報告する。

目 次

*本研究は、科学技術振興機構戦略的想像研究推進事業 (CREST) 研究領域「情報社会を支える新しい高性能情報処理技術」研究課題「検証における記述量爆発問題の構造変換による解決」の一部として遂行された。

1 はじめに

有限状態遷移系に対する形式的手法、特にモデル検査 (model checking) 手法を適用して、設計の仕様に対する妥当性を検証する方法は、近年、大きな成功をおさめている。特に、ハードウェアの設計の分野でこれは顕著である。

モデル検査の適用を試みていた初期の段階では、有限遷移系の到達可能な状態を順にたどって仕様を検証する方法が用いられていた。この方法だと、多数の状態をすべて検査するために非常に大きなメモリ空間/時間を必要とするため、現実的に検査可能な状態数はあまり大きくはなく、数百万のオーダーであった。さらに、システムを表すパラメータの数に対して、到達可能な状態数は指数的に増大するので、実際に産業界において求められる大きさのモデルを検証することは困難であった。

この状況は、記号モデル検査 (symbolic model checking) 法によって改善された。記号モデル検査法では、boolean 関数によって状態の集合を表現する。boolean 関数の操作は、2 進決定ダイアグラム (BDD - Binary Decision Diagram) と呼ばれる構造によって効率的に行われるため、必要な空間/時間量が大きく節約される。、この手法によって、状態数 10^{20} 程度が扱えるようになり、特にハードウェア設計の分野では、モデル検査が実用的なメモリ空間と時間で可能となってきた。

しかし、前述のように状態数はパラメータに対して指数的に増大する。これはしばしば「状態数爆発 (state explosion) 問題」と呼ばれる。パラメータ数が増大すれば、システムは、記号モデル検査法によっても扱いきれなくなる大きさになってしまう。このことは特にソフトウェアについてのモデル検査で大きな問題となる。

設計を対象にモデル検査を行う場合、システムの実装が設計に合致していなかったとすると、システム開発における手法としての意味をなさない。しかし、ソフトウェア開発においては、しばしば実装フェーズにおいて設計に対する変更が行われることがある。一般に、モデル化には大きな工数が必要となるため、モデルを設計変更に追随させることが、実際上困難である場合が多い。そこで、ソースコードからモデルを自動的に抽出することで、ソースコードそのものをモデル検査の対象にしようという考え方があり、最近の重要な研究対象となっている。

この場合、状態数の爆発は、ハードウェア設計を対象とする場合よりもはるかに顕著である。たとえば、100 行のコードに 32bit の int 型の変数が 10 個現れたとして、これをそのまま遷移系と考えると、状態数は $100 \times (2^{32})^{10}$、およそ 10^{100} となる。状態を効率的に表現するだけでは到底実用とならず、状態数そのものを減らすことが必要になる。この手法が抽象化と呼ばれるものである。

もちろん、ハードウェア設計を対象としたモデル検査においても抽象化は有効な手法であるが、特にソフトウェアのソースコードを対象としたモデル検査においては、抽象化は必須となる。頻繁に変更されるソースコードを検証の対象とする以上、モデルは (半) 自動的に生成されなければならない。抽象化は、モデル生成の過程で行われるものであるから、その過程に組み込まれる、有効な抽象化を行うツールが必要となる。

本報告では、以上のような観点から、抽象化ツールに焦点を当てることにした。抽象化を実現する種々のツールを、第 3 節以降の各節で 1 つずつ紹介する。ベースとしている理論はツールごとに異なるので、各節で紹介するが、基本的な部分は次の第 2 節でまとめている。

2 抽象化手法

この節では、後の節で必要となる、抽象化手法に関する一般的な事項を述べる。まず準備として、2.1 節で性質の記述に使用される時相論理の文法を、2.2 節では時相論理の意味論をまとめる。2.3 節で抽象化の一般論を述べ、2.4 節ではその理論的な背景であるガロア接続を紹介する。2.5 節では、以

上の一般論をソフトウェアモデル検査に適用する考え方を述べる。最後に、2.6 節から 2.8 節までで、代表的な抽象化手法であるデータマッピング法と述語抽象化法を紹介する。

2.1 時相論理

時相論理 **CTL*** の論理式は、以下の BNF[1] で定義される。

<atomic proposition> ::= $P_1 \mid P_2 \mid \ldots$

<state formula> ::=

<atomic proposition> | ¬<atomic proposition>

| <state formula> ∨ <state formula> | <state formula> ∧ <state formula>

| **A** <path formula> (1)

| **E** <path formula> (2)

<path formula> ::=

<state formula> (3)

| <path formula> ∨ <path formula> | <path formula> ∧ <path formula> (4)

| **X** <path formula> | **F** <path formula> | **G** <path formula> (5)

| <path formula> **U** <path formula> | <path formula> **R** <path formula> (6)

<CTL* formula> ::= <state formula> | <path formula>

時相論理で使用される記号は文献により異なっており、他の文献では、表 1に示す記号も用いられる。

CTL*の部分として定義される論理式のクラスに、以下のものがある。

ACTL* は、CTL*の BNF から、行 (2) の「| **E** <path formula>」を除いたもので定義される <state formula>と <path formula>の全体である。すなわち、「**E** の出てこない CTL*論理式」ということになる。

ECTL* は、CTL*の BNF から、行 (1) の「| **A** <path formula>」を除いたもので定義される <state formula>と <path formula>の全体である。すなわち、「**A** の出てこない CTL*論理式」ということになる。

表 1: 時相論理式に用いる記号

本報告	他の文献	
A	∀	
E	∃	
X	◯	
F	◇	<>
G	□	[]
R	**V**	

[1] ACTL*や ECTL*の定義の便宜のため、ここでは、CTL*の否定標準形 (negation normal form) の BNF を示した。通常の CTL*よりも論理式の範囲は狭いが、記述力は変わらない。

CTL (Computation Tree Logic) は、CTL*の BNF の「<path formula>」を次のように書き換えたもので定義される<state formula>の全体である。

<path formula> ::=

| **X** <state formula> | **F** <state formula> | **G** <state formula>

| <state formula> **U** <state formula> | <state formula> **R** <state formula>

すなわち、「**A**,**E** と **X**,**F**,**G**,**U**,**R** とが必ずペアになって現れる CTL*論理式」ということになる。なお、CTL の「<path formula>」の式は、CTL*の「<path formula>」の式から行 (3)、(4) を削除し、行 (5)、(6) に現れる「<path formula>」を「<state formula>」に変更して得られるものになっている。

LTL (Linear Temporal Logic) は、CTL*の BNF に現れる、行 (3) の「<state formula> |」を「<atomic proposition> | ¬<atomic proposition> |」で置き換えて得られる<path formula>の全体である。すなわち、「**A** と **E** が現れない CTL*論理式」ということになる。

なお、以下では、「$\varphi \to \psi$」を、「$\neg\varphi \vee \psi$ を否定標準形に書き直したもの」の略記として使用することがある。

2.2 Kripke 構造

時相論理 CTL*およびその部分論理式クラスのモデルは、Kripke 構造によって与えられる。まず、Kripke 構造のもとになる遷移系から述べる。

遷移系 (transition system) とは、3 つ組 (Q, R, I) であって、R は集合 Q 上の関係 ($R \subseteq Q \times Q$)、集合 I は集合 Q の空でない部分集合 ($\emptyset \neq I \subseteq Q$) になっているものである。$Q$ を**状態** (state) の集合、R を**遷移関係** (transition relation)、I を**初期状態** (initial state) の集合と呼ぶ。sRs' (すなわち、$(s, s') \in R$) のとき、状態 s から状態 s' に遷移する、という。集合 $\{\ s' \in Q \mid sRs'\ \}$ の要素数が 1 のとき、状態 s からの遷移は**決定的** (deterministic) であるという。要素数が 2 以上のときは、**非決定的** (non-deterministic) であるという。要素数が 0 のときは、その状態を**デッドロック** (deadlock) という。

自然数全体の集合 $\{0, 1, \ldots\}$ を $\mathbf{N}$ で表す。遷移系の状態の有限列 $\pi = s_0 s_1 \ldots s_{n-1}$ ($s_i \in Q$) について、その長さ $\mathrm{len}(\pi)$ を n と定める。遷移系の状態の無限列 $\pi = s_0 s_1 \ldots$ ($s_i \in Q$) については、その長さ $\mathrm{len}(\pi)$ を ∞ と定める。

遷移系の状態の有限または無限列 π が、次の条件を満たすとき、各 $i \in \mathbf{N}$ に対して $s_i R s_{i+1}$ を満たすとき、π を遷移系の**実行経路** (path) と呼ぶ。

- 各 $0 < i < \mathrm{len}(\pi)$ に対して、$s_{i-1} R s_i$。
- $\mathrm{len}(\pi) < \infty$ の場合には、最後の状態 $s_{len(\pi)-1}$ はデッドロック。

ただし、$i \in \mathbf{N}$ に対して、$i < \infty$ と約束する。

$i < \mathrm{len}(\pi)$ のとき、実行経路 π の i 番目の状態 s_i を $\pi(i)$ と書き、π の i 番目から始まる実行経路 $s_i s_{i+1} \ldots$ を π^i と書く。

例 2.1 $Q = \{a, b, c, d\};\ R = \{(a, b), (b, c), (b, d), (c, a), (c, c)\};\ I = \{a\}$ として、(Q, R, I) は遷移系である。これを、図 1のように表現する。丸が状態を表し、遷移があることを矢印で表している。初期状態は、元が空の矢印で表している。状態 a からの遷移は決定的、状態 b や c からの遷移は非決定的、状態 d はデッドロックである。列 $\pi = cabccccc \ldots$ は実行経路であり、$\pi^2 = bccccc \ldots$ である。 ∎

時相論理のモデルを定義するため、まず、**原子述語** (atomic predicate) の集合を固定し、$AP = \{P_1, P_2, \ldots, P_m\}$ とする。次のような 4 つ組 (Q, R, I, λ) を、**Kripke 構造** (Kripke structure) という。

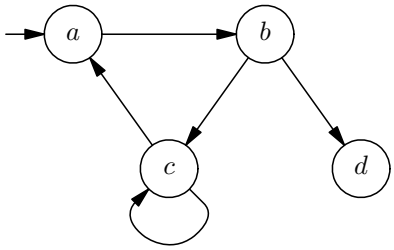


図 1: 遷移系の例

- (Q, R, I) は遷移系。
- λ は $\lambda : Q \to 2^{AP}$ なる写像。**ラベル付け写像** (labeling function) と呼ばれる。

直観的には、状態 $s \in Q$ に対し、$\lambda(s)$ は、s 上で成立する AP の述語全体を表している。

Kripke 構造 $\mathcal{M} = (Q, R, I, \lambda)$ の状態 $s \in Q$ と実行経路 π、および CTL*の state formula φ と path formula ψ の間の関係 $\mathcal{M}, s \models \varphi$ と $\mathcal{M}, \pi \models \psi$ を、次のように帰納的に定義する。ただし、P は原子述語、φ, φ_1, φ_2 は state formula、ψ, ψ_1, ψ_2 は path formula を表すものとし、また、誤解のおそれのない時には、$\mathcal{M}, s \models \varphi$ および $\mathcal{M}, \pi \models \psi$ を、それぞれ $s \models \varphi$ および $\pi \models \psi$ と略記する。

$$
\begin{array}{lcl}
s \models P & \Longleftrightarrow & P \in \lambda(s) \\
s \models \neg P & \Longleftrightarrow & P \notin \lambda(s) \\
s \models \varphi_1 \vee \varphi_2 & \Longleftrightarrow & s \models \varphi_1 \text{または } s \models \varphi_2 \\
s \models \varphi_1 \wedge \varphi_2 & \Longleftrightarrow & s \models \varphi_1 \text{かつ } s \models \varphi_2 \\
s \models \mathbf{E}\,\psi & \Longleftrightarrow & \exists \pi[\ \pi(0) = s \text{ かつ } \pi \models \psi\ ] \\
s \models \mathbf{A}\,\psi & \Longleftrightarrow & \forall \pi[\ \pi(0) = s \text{ ならば } \pi \models \psi\ ] \\
\pi \models \varphi & \Longleftrightarrow & \pi(0) \models \varphi \\
\pi \models \psi_1 \vee \psi_2 & \Longleftrightarrow & \pi \models \psi_1 \text{ または } \pi \models \psi_2 \\
\pi \models \psi_1 \wedge \psi_2 & \Longleftrightarrow & \pi \models \psi_1 \text{ かつ } \pi \models \psi_2 \\
\pi \models \mathbf{X}\,\psi & \Longleftrightarrow & \mathrm{len}(\pi) > 1 \text{ かつ } \pi^1 \models \psi \\
\pi \models \mathbf{F}\,\psi & \Longleftrightarrow & \exists k < \mathrm{len}(\pi)[\ \pi^k \models \psi\ ] \\
\pi \models \mathbf{G}\,\psi & \Longleftrightarrow & \forall k < \mathrm{len}(\pi)[\ \pi^k \models \psi\ ] \\
\pi \models \psi_1\,\mathbf{U}\,\psi_2 & \Longleftrightarrow & \exists k < \mathrm{len}(\pi)[\ \pi^k \models \psi_2 \text{ かつ } \forall j < k[\ \pi^j \models \psi_1\ ]\ ] \\
\pi \models \psi_1\,\mathbf{R}\,\psi_2 & \Longleftrightarrow & \forall k < \mathrm{len}(\pi)[\ \pi^k \models \psi_2 \text{ または } \exists j < k[\ \pi^j \models \psi_1\ ]\ ]
\end{array}
$$

state formula φ が任意の初期状態 $s \in I$ に対して $\mathcal{M}, s \models \varphi$ を満たすとき、$\mathcal{M} \models \varphi$ と書き、φ は $\mathcal{M}$で正しい、という。path formula ψ が $\pi(0) \in I$ なる任意の実行経路 π に対して $\mathcal{M}, \pi \models \psi$ を満たすとき、$\mathcal{M} \models \psi$ と書き、ψ は $\mathcal{M}$で正しい、という。

2.3 抽象化

Kripke 構造 $\mathcal{C} = (C, R, I, \lambda)$ と $\mathcal{A} = (A, \overline{R}, \overline{I}, \overline{\lambda})$ に対し、次の条件をみたす写像 $\beta : C \to A$ があるときに、$\mathcal{A}$を$\mathcal{C}$の**抽象構造** (abstract structure)、$\mathcal{C}$を $\mathcal{A}$の**具体構造** (concrete structure) という。

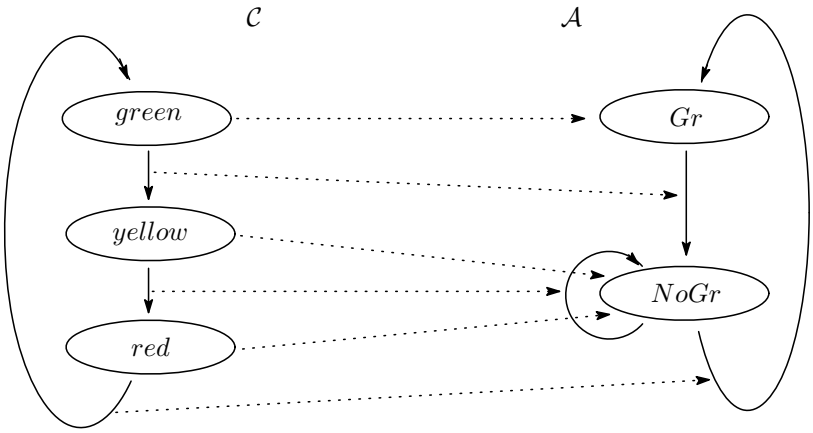


図 2: 交通信号モデルの抽象化

写像 β を**抽象化写像** (abstraction mapping) と呼ぶ。

$$\beta(I) \subseteq \overline{I} \tag{7}$$

$$c, c' \in C \text{ に対し、} \quad cRc' \implies \beta(c)\overline{R}\beta(c') \tag{8}$$

$$c \in C \text{ に対し、} \quad \lambda(c) = \overline{\lambda}(\beta(c)) \tag{9}$$

例 2.2 交通信号をモデル化する。原子述語の集合は $AP = \{isGreen\}$ とする。下の λ の定義に見るように、$isGreen$ は、「青色である」ことを示す述語である。具体構造として、Kripke 構造 $\mathcal{C} = (C, R, I, \lambda)$ を次のように定義する。

- $C = \{green, yellow, red\}$
- $R = \{(green, yellow), (yellow, red), (red, green)\}$
- $I = C$
- $\lambda(green) = \{isGreen\}, \quad \lambda(yellow) = \lambda(red) = \emptyset$

これに対して、黄色と赤の違いを無視した Kripke 構造 $\mathcal{A} = (A, \overline{R}, \overline{I}, \overline{\lambda})$ を、次のように与える。

- $A = \{Gr, NoGr\}$
- $\overline{R} = \{(Gr, NoGr), (NoGr, NoGr), (NoGr, Gr)\}$
- $\overline{I} = A$
- $\overline{\lambda}(Gr) = \{isGreen\}, \quad \overline{\lambda}(NoGr) = \emptyset$

すると、写像 $\beta : C \to A$ を $\beta(green) = Gr$、$\beta(yellow) = \beta(red) = NoGr$ と定めることによって、$\mathcal{A}$ は β を抽象化写像とする $\mathcal{C}$ の抽象構造となる。(図 2) ∎

この例でもわかるように、一般に、抽象構造では非決定的な遷移がある。

応用の場面では、具体的な遷移系と検証すべき性質から、抽象化写像 β をうまく与えて $\mathcal{A}$を作ることが課題となる。この場合、検証すべき性質を表現できるように述語集合 AP を定めると、ラベル付け写像 λ は自然に決まる。次に、A と $\beta : C \to A$ を決めるが、β は条件

$$c, c' \in C \text{ に対し、} \quad \beta(c) = \beta(c') \implies \lambda(c) = \lambda(c') \tag{10}$$

を満たす全射となるようにする。このようにすれば、次の $\overline{\lambda}$ の定義が well-defined となり、条件 (9) を満たす。

$$\overline{\lambda}(\beta(c)) = \lambda(c) \tag{11}$$

初期状態集合 $\overline{I}$ は、通常 $\overline{I} = \beta(I)$ と定める。

遷移関係 $\overline{R}$ は、式 (8) を満たすように定めなければならない。$a\overline{R}a' \stackrel{\text{def}}{\iff} \exists c \exists c' [\, a = \beta(c) \wedge a' = \beta(c') \wedge cRc' \,]$ と定義できれば、偽反例を最小にするという観点からは最善であり、これを実現するツールもある。場合によっては、遷移関係を決定するための計算量の観点から、右辺が成り立つ場合に必ず $a\overline{R}a'$ となるように定義することもある。計算時間の大半が遷移関係の決定のために費やされることが多く、抽象化を応用する際のポイントの 1 つとなっている。

抽象化の方法の妥当性は、次の定理で保証される。

定理 2.3 Kripke 構造 $\mathcal{A}$ が Kripke 構造 $\mathcal{C}$ の抽象構造であるとき、ACTL*論理式 φ に対し、$\mathcal{A} \models \varphi$ ならば $\mathcal{C} \models \varphi$ である。 ∎

この定理から、検証したい性質を ACTL*で表現して、抽象構造のモデル検査によって正しいことを確かめれば、具体構造、すなわち実際のプログラムもその性質を持つことがわかる。この事実を、$\mathcal{A}$ は $\mathcal{C}$ に対して**健全** (sound) な抽象化である、とか、性質を**弱い意味で保存する** (weakly preserves) と表現する。

一般には定理 2.3の逆は成立しない。つまり、抽象構造におけるモデル検査で検証したい性質が否定されたとしても、実際のプログラムではその性質は成り立っているかもしれない。しかしこのことは、必ずしも抽象化法が無力であることを意味しない。抽象構造 $\mathcal{A}$のモデル検査で、ACTL*の性質が否定された場合には、$\mathcal{A}$の実行経路 $\bar{\pi}$ で、検証したい性質の**反例** (counterexample) となるものがみつかったことになる。そこで、この反例を検討して、具体構造 $\mathcal{C}$の実行経路 π で、$\bar{\pi} = \beta(\pi)$ となるものが存在するかどうかを調べる。存在すれば、$\mathcal{C}$でもこの性質が否定されたことになる。このような反例 $\bar{\pi}$ を、**実反例** (actual counterexample) と呼ぶ。そうでない場合、この反例を、**偽反例** (spurious counterexample) と呼ぶ。この場合には、偽反例を排除するように抽象構造に変更を加えることが検討される。

例 2.4 例 2.2において、ACTL*論理式 $\mathbf{AGF}(\neg isGreen)$ を φ_1 とすると、$\mathcal{A} \models \varphi_1$ であるから、定理 2.3より、$\mathcal{C} \models \varphi_1$ であることがわかる。(この例では、直接 $\mathcal{C}$を調べても簡単ではあるが。)

一方、ACTL*論理式 $\mathbf{AGF}(isGreen)$ を φ_2 とすると、$\mathcal{C} \models \varphi_2$ である。しかし、抽象構造 $\mathcal{A}$では、$NoGr\ \overline{R}\ NoGr$ であるから、実行経路として、$NoGr, NoGr, NoGr, \ldots$ というものがとれてしまう。したがって、$\mathcal{A} \not\models \varphi_2$ であり、この実行経路は偽反例である。 ∎

健全な抽象化について述べてきたが、写像 $\beta : C \to A$ について、次に示すもっと強い条件を考えることもある。

$$\beta\text{は全射。} \tag{12}$$

$$\beta(I) = \overline{I} \tag{13}$$

$$c, c' \in C \text{ に対し、} cRc' \iff \beta(c)\overline{R}\beta(c') \tag{14}$$

$$c \in C \text{ に対し、} \lambda(c) = \overline{\lambda}(\beta(c)) \tag{15}$$

β がこれらを満たす場合には、ACTL*論理式 φ に対して、$\mathcal{A} \models \varphi \iff \mathcal{C} \models \varphi$ となる。このことを、**完全** (complete) な抽象化、とか、性質を**強い意味で保存する** (strongly preserves) と表現する。しかし、このような抽象化では、$\mathcal{A}$の状態数が $\mathcal{C}$の状態数に比べて十分小さくならず、あまり実用的でないことが多い。

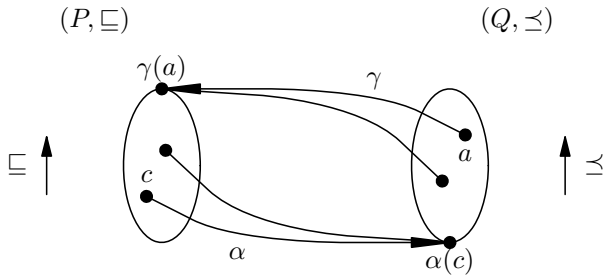


図 3: ガロア接続

2.4 ガロア接続

$(P, \sqsubseteq)$ と $(Q, \preceq)$ を順序集合とする。写像の組 $\alpha : P \to Q$, $\gamma : Q \to P$ が次の条件をみたすとき、(α, γ) を**ガロア接続** (Galois connection) という (図 3)。

- α と γ は、順序を保存する。すなわち、$c, c' \in P$ に対して $c \sqsubseteq c' \implies \alpha(c) \preceq \alpha(c')$、$a, a' \in Q$ に対して $a \preceq a' \implies \gamma(a) \sqsubseteq \gamma(a')$。
- $c \in P$ に対して、$c \sqsubseteq \gamma \circ \alpha(c)$
- $a \in Q$ に対して、$\alpha \circ \gamma(a) \preceq a$

上の 3 条件は、次のようにいいかえても同じである。

- α と γ は、順序を保存する。
- $a \in Q$ と $c \in P$ に対して、$\alpha(c) \preceq a \iff c \sqsubseteq \gamma(a)$

このとき、$a \in Q$ と $c \in P$ に対して、次の関係が成り立つ。

- $\gamma(a) = \max\{\, c \in P \mid \alpha(c) \preceq a \,\}$
- $\alpha(c) = \min\{\, a \in Q \mid c \sqsubseteq \gamma(a) \,\}$

ガロア接続の合成は、ガロア接続である。すなわち、順序集合 P, Q, Q' に対し、(α, γ) が、P と Q の間のガロア接続で、(α', γ') が、Q と Q' の間のガロア接続ならば、$(\alpha' \circ \alpha, \gamma \circ \gamma')$ は、P と Q' の間のガロア接続である。

ガロア接続 (α, γ) が、さらに、次の条件をみたすとき、この組を**ガロア挿入** (Galois insertion) という。

- $a, a' \in Q$ に対し、$\gamma(a) \sqsubseteq \gamma(a') \implies a \preceq a'$

上の条件は、次のように書き直すこともできる。

- $a \in Q$ に対し、$\alpha \circ \gamma(a) = a$

特に、P と Q が冪集合の場合、すなわち、$(P, \sqsubseteq) = (2^C, \subseteq)$、$(Q, \preceq) = (2^A, \subseteq)$、の場合を考える。このときには、写像 $\beta : C \to A$ を用いて次のように定義した (α, γ) はガロア接続になる。

- $X \in 2^C$ に対し、$\alpha(X) = \{\, \beta(x) \mid x \in X \,\}$
- $Y \in 2^A$ に対し、$\gamma(Y) = \{\, x \in P \mid \beta(x) \in Y \,\}$

β が全射の場合には、(α, γ) は、ガロア挿入となる。

Kripke 構造 (C, R, I, λ) とその抽象構造 $(A, \overline{R}, \overline{I}, \overline{\lambda})$ をとり、抽象化写像を β とする。$(2^C, \subseteq)$ と $(2^A, \subseteq)$ の間の上記のガロア接続 (α, γ) を考える。$f : 2^C \to 2^C$ を $f(X) = X \cup \{\, c' \in C \mid \exists c \in X\, [\, cRc' \,] \,\}$ で、$g : 2^A \to 2^A$ を $g(Y) = Y \cup \{\, a' \in A \mid \exists a \in X [\, a\overline{R}a' \,] \,\}$ で定義する。β が抽象化写像であることから、$\alpha(f(X)) \subseteq g(\alpha(X))$ となる。$X \subseteq f(X)$ と $X \subseteq X' \implies f(X) \subseteq f(X')$ とが成り立つので、$X \in 2^C$ に対して、X を含む不動点 $X_0 \in 2^C$、すなわち $X \subseteq X_0$, $f(X_0) = f(X_0)$ となるものが存在する。このような X_0 のうち、最小のものを $L(f, X)$ と書く。$L(f, X)$ は、X から R によって到達できる状態の集合である。同様に 2^A のなかで $L(g, Y)$ が定義できる。このとき、

```
        extern int x;

L1:     while (x != 0) {
L2:         if (x > 0) {
L3:             x--;
            }else if (x < 0) {
L4:             x = -x;
            }
        }
L5:     ;
```

図 4: サンプルソース

$L(f, X) \subseteq \gamma(L(g, \alpha(X)))$ が導かれる。したがって、$\alpha(I) \subseteq \overline{I}$ に注意して、$L(f, I) \subseteq \gamma(L(g, \overline{I}))$ となる。これは、具体構造における到達可能状態の集合を、抽象構造を調べることによって、上から評価することができることを意味している。

2.5 プログラムへの適用

前節までの議論は、一般の遷移系についてのものである。抽象化ツールは、C や java などの言語で記述されたプログラムを対象とするものが多い。この節では、プログラムに対して前節までで与えた枠組みを適用する方法について述べる。

遷移系

C や java などの言語で書かれたプログラム P は、次のように、遷移系と考えることができる。まず、各実行文の直前にラベルを置き、ラベルの集合を PC とする。また、P が持つ変数を $v_1, \ldots, v_n$ として、v_i の型に属する値の集合を D_i とする。遷移系の状態集合は、$Q = PC \times D_1 \times \ldots \times D_n$ とする。遷移関係 R は、ラベル pc の位置で、変数の値が $d_1, \ldots, d_n$ であったとき、実行文によって変数の値が $d_1', \ldots, d_n'$ となり、次のラベルが pc' となる場合に、$(pc, d_1, \ldots, d_n) R (pc', d_1', \ldots, d_n')$ であるとする。初期状態は、プログラム開始時の状態である。

複数のプロセスが並行して動作する場合を取り扱うときには、各プロセス j が実行しうる文に対するラベルの集合を PC_j として、$Q = PC_1 \times \ldots \times PC_m \times D_1 \times \ldots \times D_n$ とする。

例 2.5 図 4に示す C 言語のソースに対応する遷移系 (Q, R, I) を考える。Z を、`int` が表す範囲、たとえば $[-2^{31} .. 2^{31} - 1]$ とする。状態集合は $Q = \{L1, L2, L3, L4, L5\} \times Z$ となり、初期状態の集合は $I = \{ (L1, i) \mid i \in Z \}$ である。また、遷移関係 R については、たとえば $(L1, 1)\ R\ (L2, 1)\ R\ (L3, 1)\ R\ (L1, 0)\ R\ (L5, 0)$ が成り立つ。 ∎

Kripke 構造

遷移系を Kripke 構造 $\mathcal{C} = (C, R, I, \lambda)$ に拡張するためには、原子述語の集合を定める必要がある。プログラムが満たすことを検証したい性質があるので、原子述語の集合は、これらの性質を記述することができるように定めればよい。

例 2.6

例 2.5のプログラムにおいて、次のことを検証したいとする。

(1)「いつかは変数 x の値が 0 以上になる。」

(2)「変数 x の値がひとたび 0 以上になれば、その後 0 以上の値であることを保ったままいつかは while ループを抜ける。」

この場合、P_1 = "$x \geq 0$"、P_2 = "$pc = L5$" として、$AP = \{P_1, P_2\}$ とする。すると、(1) は、$\mathbf{AF}P_1$、(2) は $\mathbf{AG}(P_1 \rightarrow (P_1 \,\mathbf{U}\, P_2))$ と表すことができる。

ラベル付け写像 λ は、自然に決めることができる。たとえば、$\lambda((L1, 5)) = \{P_1\}$, $\lambda((L5, 0)) = \{P_1, P_2\}$ などとする。 ∎

抽象化

プログラムから作られた Kripke 構造の状態集合は、有限であっても非常に大きな集合となりうる。また、データの範囲に制限をつけられずに、無限集合と考えることもある。それに対して抽象構造を十分小さく作ることによって、モデル検査を適用できるようにすることが、抽象化の目的である。

多くの抽象化ツールで採用されている抽象化の方法に、**データマッピング** (data mapping) 法と**述語抽象化** (predicate abstraction) 法がある。これらについて、2.6節と 2.7節で述べる。

抽象構造の状態数は十分小さくなければならないが、これは、具体構造の状態のうち、検証しようとする性質に関係のない状態を、いわば「捨てる」ことによって実現される。ところで、ひとつのプログラムに対して、検証したい性質は複数 (おそらく数十、数百と) あることが普通であり、ほとんどの具体構造の状態は、これらの性質のどれかには関係している。だから、ひとつの抽象化によってこれらの性質すべてを検証しようとするのでは、状態の数を小さくすることはできない。そこで、検証したい性質ごとに抽象構造を構成することが必要になってくる。したがって、できる限り抽象構造が自動的に作成されることが望まれる。

性質の記述例

定理 2.3を適用するためには、検証する性質は ACTL*論理式で表現できなければならない。一般に検証を行いたいような性質のうちの多くのものが、ACTL*論理式で表現できる。以下に例を示す。

例 2.7 「(無限ループにならずに) いつかはループの直後のラベル L に到達する。」$\mathbf{AF}(\mathrm{pc} = \mathrm{L})$ ∎

例 2.8 「変数 x の値は、負になることはない。」$\mathbf{AG}(\mathrm{x} \geq 0)$

これは、いわゆる**安全性** (safety) の例である。安全性の性質は、反例が有限の実行経路で与えられるという特徴がある。

構造 $\mathcal{M}$ で $\mathbf{AG}\varphi$ が成り立つとき、φ を、$\mathcal{M}$の**不変式** (invariant) と呼ぶ。 ∎

例 2.9 「資源割り当て要求が出されたら、いつかは資源が割り当てられる。」$\mathbf{AG}(\mathrm{req} = \mathrm{true} \rightarrow \mathbf{F}(\mathrm{acquire} = \mathrm{true}))$

これは、いわゆる**活性** (liveness) の例である。活性の性質は、反例が無限の実行経路で与えられるという特徴がある。 ∎

例 2.10 「ずっと受付可能な状態ならば、いつかは受付を行う。」$\mathbf{A}(\mathbf{GEX}\mathit{accept} \rightarrow \mathbf{F}\mathit{accept})$

この性質は、**弱い公平性** (weak fairness) と呼ばれる。これは、$\mathbf{AF}(\mathbf{AX}(\neg \mathit{accept}) \vee \mathit{accept})$ と同値であるから、ACTL*論理式である。 ∎

例 2.11 「無限回受付可能になるなら、いつかは受付を行う。」$\mathbf{A}(\mathbf{GFEX}\mathit{accept} \rightarrow \mathbf{F}\mathit{accept})$

この性質は、**強い公平性** (strong fairness) と呼ばれる。これは、$\mathbf{AF}(\mathbf{GAX}(\neg \mathit{accept}) \vee \mathit{accept})$ と同値であるから、ACTL*論理式である。 ∎

ただし、ACTL*論理式では表現できない性質もたくさんある。たとえば、「ある性質が公平性をもてば別のある性質が成り立つ」という性質は、一般には ACTL*論理式では表現できない。

2.6 データマッピング

データマッピング法 [11] は、プログラムに現れる各変数ごとに抽象を行う方法である。

変数 $v_1, \ldots, v_n$ とラベルの集合 PC を持つプログラム P から、前節の方法で与えた具体構造 $\mathcal{C} = (C, R, I, \lambda)$ を考える。構造 $\mathcal{A} = (A, \overline{R}, \overline{I}, \overline{\lambda})$ と抽象化写像 β を構成する。

2.3節で述べたように、抽象化を定めるためには、状態集合 A、条件 (10) を満たす全射の抽象化関数 β、および遷移関係 $\overline{R}$ を決める。データマッピング法での A と β の取り方を述べる。

まず、各変数 v_i の値の集合 (具体データドメイン) を D_i として、抽象データドメイン $\overline{D_i}$ と、マッピング $h_i : D_i \to \overline{D_i}$ を指定する。そして、状態集合は $A = PC \times \overline{D_1} \times \ldots \overline{D_n}$ と定める。抽象化写像 $\beta : C \to A$ は、$\beta(pc, d_1, \ldots, d_n) = (pc, h_1(d_1), \ldots, h_n(d_n))$ である。この β が条件 (10) を満足するように、h_i を決める必要がある。

$\overline{D_i}$ と h_i としては、次のものがよく用いられる。

- 1 点につぶす関数。$\overline{D_i} = \{p\}$, $h_i(d_i) = p$
  変数 v_i が検証すべき性質に全く関係ないときに用いられる。
- 抽象化を行わない関数。$\overline{D_i} = D_i$, $h_i(d_i) = d_i$
  D_i が小さな集合で、すべての値が、示すべき性質に対して独自の意味を持つ場合などに用いられる。
- ある範囲に限定する関数。D_i が離散的な順序集合であるとき、$m, M \in D_i$, $m \leq M$ として、$\overline{D_i} = \{small, m, \ldots, M, large\}$,

$$h_i(d_i) = \begin{cases} small & (d_i < m) \\ d_i & (m \leq d_i \leq M) \\ large & (d_i > M) \end{cases}$$

  特に、整数型の変数に対し、$m = M = 0$ としたものは、よく用いられる。このときの D_i を、$\{NEG, ZERO, POS\}$ と書くことにする。
- 整数型 (の部分型) の変数に対し、k を法とする剰余を対応させる関数。

例 2.12 例 2.6の Kripke 構造に対して、データマッピングを適用してみる。関係する変数は x だけである。述語集合は $\{x \geq 0,\ pc = L5\}$ であった。

マッピングは、前述の $\{NEG, ZERO, POS\}$ をとる。条件 (10) を満たすことは簡単に確認できる。遷移関係 $\overline{R}$ の作り方も、この例では自明であり、図 5のようになる。(一般には、if 文などの条件式に対して、抽象データドメインの各要素が満たす条件が充足可能かどうかを決定することが必要となり、定理証明器を使用することになる。)

例 2.6であげた性質を検証してみると、$\mathcal{A} \models \mathbf{AF}(x \geq 0)$ は成り立つ。したがって、$\mathcal{C} \models \mathbf{AF}(x \geq 0)$ が結論できる。一方、$\mathcal{C} \models \mathbf{AG}(P_1 \to P_1 \,\mathbf{U}\, P_2)$ であるが、$\mathcal{A} \models \mathbf{AG}(P_1 \to P_1 \,\mathbf{U}\, P_2)$ は成り立たない。(L1,POS) $\overline{R}$ (L2,POS) $\overline{R}$ (L3,POS) $\overline{R}$ (L1,POS) ... という偽反例が存在するからである。

∎

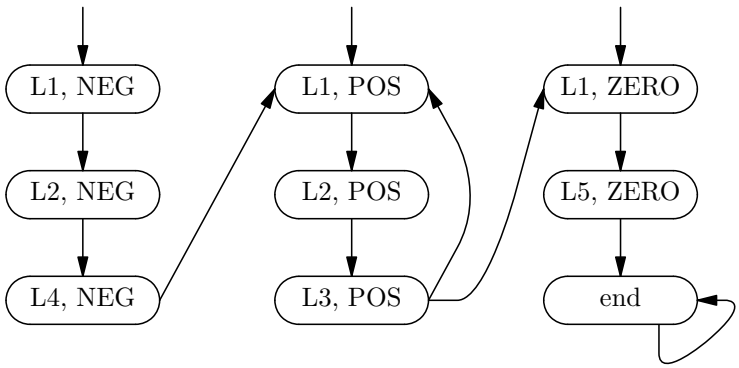


図 5: サンプルソースの抽象構造

2.7 述語抽象化

データマッピング法では、変数ごとにしか抽象化ができない。したがって、複数の変数の組み合わせが問題になる場合には、適用できない。述語抽象化法 [20] では、一般的な抽象化が可能である。

前節と同じように、変数 $v_1, \ldots, v_n$ を持つプログラム P から構成された具体構造 $\mathcal{C}$ を考え、状態集合を $C = PC \times D_1 \times \ldots \times D_n$ とする。C 上で、m 個の述語 $\varphi_1(pc, d_1, \ldots, d_n), \ldots, \varphi_m(pc, d_1, \ldots, d_n)$ を考える。ただし、原子述語はすべてこれらの中に現れるものとする。すなわち、

$$AP \subseteq \{\varphi_1, \ldots, \varphi_m\} \tag{16}$$

となるようにする。

C の各状態は、これらの述語のどれを成り立たせるか、で分類できる。述語抽象化による抽象構造 $\mathcal{A}$は、述語の成り立たせ方が同じであるものを同一視したものである。

前節と同じように、状態集合 A と抽象化関数 β の作り方を述べることにする。状態集合は、$A = \{\ (pc, b_1, \ldots, b_m) \mid pc \in PC, b_i \in \{0,1\}\quad (i = 1, \ldots, m)\ \}$ である。たとえば、$m = 2$ として、$(pc, 1, 0) \in A$ は、「$\mathcal{C}$の状態のうち、ラベル pc の位置にいて、φ_1 は成り立っていて、φ_2 は成り立っていないようなものたち」を表したものである。次に、$i = 1, \ldots, m$ に対して b_i を

$$b_i = \begin{cases} 1 & \varphi_i(pc, d_1, \ldots, d_n) \text{ が成り立つとき。} \\ 0 & \text{そうでないとき。} \end{cases}$$

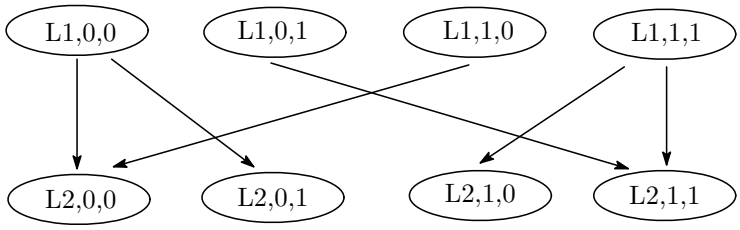
と定めるとき、抽象化写像 $\beta : C \to A$ は、$\beta(pc, d_1, \ldots, d_n) = (pc, b_1, \ldots, b_m)$ で定義される。条件 (16) により、条件 (10) が成立している。

遷移関係の計算について、例で説明する。2 つの整数型の変数 x と y を持つ C プログラム P に対し、2 つの述語 $\varphi_1 =$「$x + y \geq 0$」と $\varphi_2 =$「$x \geq 0$」とを使った述語抽象化を考える。 P に、図 6(a) に示すコードが現れるとして、この部分に対応する遷移関係を計算してみる。抽象構造の状態 $a' = (L2, 1, 1)$ を考える。$(L1, b_1, b_2)$ (ただし、$b_1, b_2 \in \{0,1\}$) の形の状態 a のそれぞれが、$a\overline{R}a'$ となるかどうかを決定したい。このために、**最弱前条件** (weakest precondition) という概念を準備する。

一般に、ある実行文の直後における条件 ψ に対し、その実行文の直前における条件 μ のうち、$\mu \Rightarrow \psi$ が成り立つもののなかでもっとも弱いものを、その実行文に関する ψ の最弱前条件という。たとえば、実行文 `x=x-1` に関する条件 $x \geq 0$ の最弱前条件は $x \geq 1$ である。実行文が代入文 `x=`e の場合には、条件 ψ の最弱前条件は、ψ の中の `x` を e に置き換えることによって得られる。

さて、状態 a' は、具体構造の状態のうち、L2 の位置で、条件 $x + y \geq 0 \wedge x \geq 0$ を満たすものに

```
L1: x = x - y;
L2:
```



(a) (b)

図 6: 遷移関係の計算

対応する。この条件の文 `x=x-y` に関する最弱前条件は、x を $x-y$ に置き換えて得られる、

$$x \geq 0 \wedge x - y \geq 0 \tag{17}$$

である。$a = (L1, b_1, b_2)$ から a' への遷移があることは、a が表す条件と条件 (17) との連言[2]が充足可能であることと同値である。たとえば $a_{00} = (L1, 0, 0)$ が表す条件は $x + y < 0 \wedge x < 0$ であるが、これと条件 (17) との連言 $x + y < 0 \wedge x < 0 \wedge x \geq 0 \wedge x - y \geq 0$ は、充足可能でない。したがって、$a_{00}\overline{R}a'$ ではない。一方、$a_{01} = (L1, 0, 1)$ が表す条件 $x + y < 0 \wedge x \geq 0$ と条件 (17) との連言 $x + y < 0 \wedge x \geq 0 \wedge x \geq 0 \wedge x - y \geq 0$ は、たとえば $(x, y) = (1, -2)$ として充足可能である。したがって、$a_{01}\overline{R}a'$ である。このようにして、すべての組み合わせについて充足可能性を調べることによって、遷移関係を決定することができる。図 6(b) に結果を示す。

以上で見たように、遷移関係の決定には、充足可能性の検査が必要となる。述語抽象化をサポートするツールは、定理証明器を使用して充足可能性をテストしている場合が多い。上の例でもわかるように、定理証明器の呼び出し回数は、使用する述語の数に対して指数関数的に増大する。そこで、ツールの実装では、呼び出し回数を押さえるための最適化が必要となる。

なお、2.6節で説明したデータマッピングは、述語抽象化の特別な場合と考えることができる。変数 v_i に対するデータマッピング関数 $h_i : D_i \to \overline{D_i}$ $(i = 1, \ldots, n)$ による抽象化は、$|\overline{D_i}|$ 個の述語 $h_i(v_i) = \overline{d}$ $(\overline{d} \in \overline{D_i})$ による抽象化と考えることができるからである。

2.8 述語抽象化法による検証の手順

前節で述語抽象化法の原理を述べた。この原理を用いた検証は、以下のような手順を踏んで行うことになる [10]。

1. 述語 (の集合) を選択し、抽象構造を構成する。
2. 抽象構造で、検証したい性質が成立するかどうかモデル検査を行う。成立していれば、検証は終了 (性質は成り立つ)。
3. そうでない場合には、得られた反例が実反例か偽反例かを判定する。実反例ならば、検証は終了 (性質は成り立たない)。
4. 偽反例だった場合には、その偽反例を現れなくするような述語を追加し、手順 1 に戻る。

述語抽象化法に基づいたツールは、上の各ステップ (およびループそのもの) の作業を支援するものになる。任意の性質について上の手順を完全に自動化するツールを作ることは不可能である。ツールによっては、検証する性質のパターンを限ってできるだけ自動化を図るものもある。また、多くの性質の検証に対応するために、コメントの付加など、ソース以外の情報を要求するものもある。

[2] 論理式を ∧(かつ) で結んだもの。conjunction。

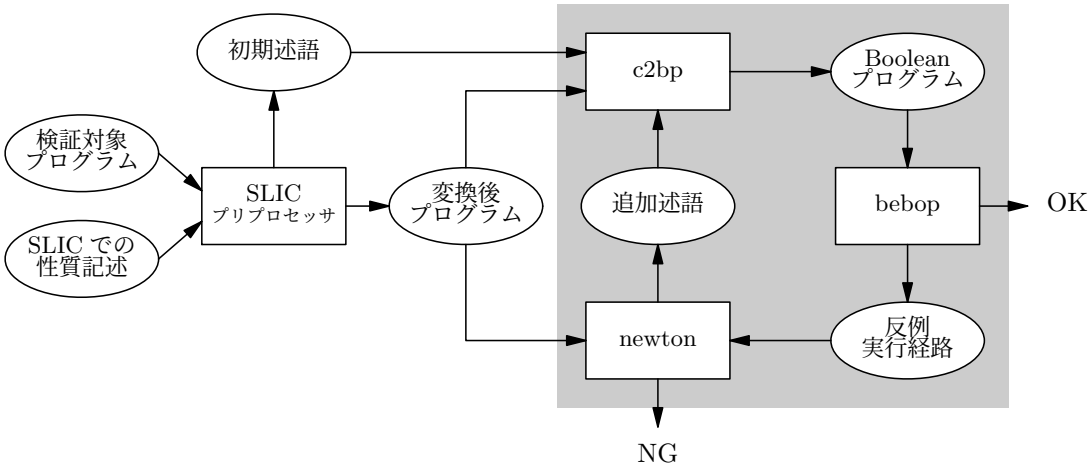


図 7: ツールキット SLAM の構成

3 SLAM

3.1 SLAM の要約

名前	SLAM
開発元	Microsoft Research
主な開発者	Thomas Ball, Sriram K. Rajamani
web ページ	`http://research.microsoft.com/SLAM/`
ツールの入手	非公開
動作環境	MS Windows
特長	C ソースコードを対象。「API の使用方法が正しいかどうか」を自動的に検査することが目標。

3.2 ツール SLAM の概要

SLAM[3] は、与えられた C のソースにおいて、「API の使用方法が正しいか」どうかをモデル検査法に基づいて検証するためのツール群である。

SLAM の構成を図 7に示す。

(1) 仕様記述言語 SLIC (Specification Language for Interface Checking)

SLIC は、C ソースコードが持つべき性質を記述するための言語である。SLIC で記述できる性質は、安全性のみである。

SLIC 言語を理解するプリプロセッサが用意されている。これに C ソースプログラム P と SLIC による検証したい性質の記述を与えると、プリプロセッサは、変換した C プログラム P' を出力する。P' では、検証したい性質が満たされないとき、そしてその時に限り ERROR というラベルに到達するようになっている。たとえば SLIC で記述するのが「ロックの獲得と解放は、かならず対になって呼び出される」という性質であれば、P' では、ロックの獲得が 2 回連続起こるようなときには ERROR ラベルに到達するわけである。そこで、性質を検証するためには、P' が ERROR ラベル

```
typedef struct cell {
  int val;
  struct cell* next;
} *list;

list partition(list *l, int v) {
  list curr, prev, newl, nextCurr;

  curr = *l; //.....................

  prev = NULL; //...................

  newl = NULL; //...................
  while (curr != NULL) { //..........

    nextCurr = curr->next; //.........
    if (curr->val > v) { //...........

      if (prev != NULL) { //..........

        prev->next = nextCurr; //.....
      }
      if (curr == *l) { //............
        *l = nextCurr; //..............
      }
      curr->next = newl; //...........
L:    newl = curr; //.................
    }else {

      prev = curr; //.................

    }
    curr = nextCurr; //..............

  }

  return newl;
}
```

(a) C ソース

```
void parition() {
  bool {curr==NULL}, {prev==NULL};
  bool {curr->val>v}, {rev->val>v};

  {curr==NULL}  = unknown();
  {curr->val>v} = unknown();
  {prev==NULL}  = true;
  {prev->val>v} = unknown();
  skip;
  while (*) {
    assume(!{curr==NULL});
    skip;
    if (*) {
      assume({curr->val>v});
      if (*) {
        assume(!{prev==NULL});
        skip;
      }
      if (*) {
        skip;
      }
      skip;
L:    skip;
    }else {
      assume(!{curr->val>v});
      {prev==NULL} = {curr==NULL};
      {prev->val>v} = {curr->val>v};
    }
    {curr==NULL} = unknown();
    {curr->val>v} = unknown();
  }
  assume({curr==NULL});
}
```

(b) Boolean プログラム

出典: 文献 [2]

図 8: リスト分割の例

に到達するかどうかをモデル検査すれば良いことになる。

SLAM で想定している利用形態は、プログラムごとに固有の性質を検証することではなく、上のロックの例のように、API セット (たとえば排他制御用の一連の API) の、「正しい使い方」が満たされているかどうかを検証する、というものである。SLIC 記述はプログラムの検証者が個々に書くのではなく、API 作成者が提供することになる。

以下の (2) から (4) は、「2.8 述語抽象化法による検証の手順」で述べた方法に従った抽象化を行うためのツール群である。

(2) 抽象化ツール c2bp

プログラム P' に対し、与えられた述語の集合を用いて抽象構造を作成するツールである。抽象構造は、Boolean プログラムと呼ばれるもので実現されている。このため、c2bp (C to Boolean Program) の名前がついている。Boolean プログラムでは、すべての変数が bool 型 ($\{true, false\}$ の値をとる) である。

詳細については「3.3 SLAM での抽象化の方法」で述べるが、たとえば図 8(a) の C ソースを c2bp に与えると、同図 (b) のような Boolean プログラムが出力される。ただし、図では c2bp の出力そのままではなく、多少見やすさのために書き換えを行ったものが示されている。なお、`{curr==NULL}` のように、式を中括弧で囲んだものは、1 つの変数を表している。また、条件式の位置の`*`は、非決

定的に true または false に評価される。関数 unknown() は、true または false を非決定的に返す。構文 assume(< 式 >) は、モデル検査において、< 式 > が成立しない実行経路を実質的に無視できるようにするものである。

はじめて c2bp が呼び出されるときには、述語の集合は、性質を記述するのに必要なもの全体で、SLIC プリプロセッサによって与えられる。2 回目以降は、直前に実行された newton が (詳細化のための) 述語の集合を与える。

(3) モデル検査器 bebop

c2bp が生成した Boolean プログラムに対するモデル検査を行うツールである。ERROR ラベルに到達する実行経路があるかどうかを検査する。そのような実行経路がなければ、検証全体が「成功」で終了する。ERROR ラベルに到達する実行経路があった場合には、その実行経路が newton によって解析される。

(4) 詳細化ツール newton

newton への入力は、C プログラム P' と、bebop が見つけた ERROR ラベルに到達する実行経路 (性質に対する反例)$\overline{\pi}$ である。newton は、まず、P' にもその反例に対応する反例が存在するかどうかを調べる。もし存在すれば、P' における反例を出力し、検証全体が「失敗」で終了する。存在しない場合、$\overline{\pi}$ は偽反例であったことになる。newton は、この場合、この偽反例を排除できるような述語を構成し、出力する。

3.3 SLAM での抽象化の方法

SLAM で、抽象化の中心となるのは、C ソースと述語集合から Boolean プログラムを生成する c2bp である。基本的な考え方は、最弱前条件を使って、定理証明器 (c2bp では、Simplify[15, 37] と Vampyre[39] が用いられている) によって、抽象構造の遷移を計算していくというものである。

ポインタ

文 a に関する条件 ψ の最弱前条件を $WP(a, \psi)$ と書くことにする。C 言語にはポインタがあるために、代入文に関する最弱前条件 $WP(\texttt{x=}e, \psi)$ が、$\psi[e/x]$ で得られるとは限らない。たとえば、ポインタ p が変数 x を指していることがわかっていれば、$WP(\texttt{x=3}, *p > 5)$ は「$*p > 5$」ではなく、「false」である。

そこで、現れる変数のすべての組み合わせに関して、別名になっているかどうかの場合を尽くして最弱前条件を作ることになる。上の例では、p が x を指す場合とそうでない場合にわけて、$(\&x = p \wedge 3 > 5) \vee (\&x \neq p \wedge *p > 5)$ が、求める $WP(\texttt{x=3}, *p > 5)$ となる。一般に ϕ に k 個の変数が現れるとき、この方式をそのまま使うと、最弱前条件は 2^k 個の論理式の選言[3]となる。c2bp ではポインタ解析を行うことで、場合の数を減らすようにしている。

[3] 論理式を $\vee$(または) で結んだもの。disjunction。

代入文の処理の実際

C ソースでの代入文に対する、Boolean プログラムでの対応する文の構成法を以下に述べる。基本的な考え方は「2.7 述語抽象化」で記したとおりである。

抽象化用の述語を $\varphi_1, \ldots, \varphi_n$ とする。Boolean プログラムでは、$\varphi_1, \ldots \varphi_n$ に対応する Boolean 変数 $b_1, \ldots, b_n$ が使用される。

各 $i (i = 1, \ldots, n)$ に対して、b_i または $\neg b_i$ を選択して n 個からなる連言を作ったものを cube と呼ぶ。たとえば $n = 3$ として、$b_1 \wedge \neg b_2 \wedge \neg b_3$ は cube である。条件 ψ に対して、cube たちの選言 c のうちで、$\mathcal{E}(c) \rightarrow \psi$ が成り立つもっとも大きいものを $\mathcal{F}(\psi)$ と書く。ただし、$\mathcal{E}(c)$ は、c に現れる b_i たちを φ_i たちで置き換えたものである。

C ソースでの代入文 `x=`e は、次の Boolean プログラムに変換される:

$$
\begin{aligned}
b_1, \ldots, b_n = \\
&\texttt{choose}(\mathcal{F}(WP(\texttt{x} = e, \varphi_1)),\ \mathcal{F}(WP(\texttt{x} = e, \neg\varphi_1))), \\
&\ldots, \\
&\texttt{choose}(\mathcal{F}(WP(\texttt{x} = e, \varphi_n)),\ \mathcal{F}(WP(\texttt{x} = e, \neg\varphi_n))),
\end{aligned}
$$

ただし、n 個の代入は並列に実行される。ここで、関数 `choose` は、次のように定義される:

```
bool choose(bool pos, bool neg) {
    if (pos) { return true; }
    if (neg) { return false; }
    return unknown();
}
```

条件文の処理の実際

条件文は、次のように処理される。$\neg\mathcal{F}(\neg\psi)$ を $\mathcal{G}(\psi)$ と書く。C ソースが、`if` (ψ) *then-part* `else` *else-part* だとすると、対応する Boolean プログラムは、次のようになる。

```
if (*) { assume(G( ψ )); then-part }
else   { assume(G(¬ψ)); else-part }
```

while 文などは、if 文と goto 文の組み合わせとして処理することができる。

3.4 適用例

SLAM 開発プロジェクトチームは、Microsoft Windows のデバイスドライバに対して SLAM を適用している。5 つのデバイスドライバ (開発中のものも含む) についてロック獲得/解放に関する API の使われ方、および、割込要求パケット処理の正しさを検証した結果が、表 2に報告されている。

4 BLAST

4.1 BLAST の要約

名前	BLAST (Berkeley Lazy Abstraction Software Verification Tool)
開発元	カリフォルニア大学バークレー校 (UCB)
主な開発者	Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Gregoire Sutre, Adam Chlipala
web ページ	`http://www-cad.eecs.berkeley.edu/~rupak/blast/`
ツールの入手	web ページからダウンロード可能 (v1.0)
動作環境	Linux と MS Windows でテスト済み
特長	C 言語ソースコードの安全性性質を述語抽象化法でモデル検査するツール。「lazy abstraction」という手法を用いて抽象化-検証-詳細化ループを効率化する。

4.2 ツール BLAST の概要

BLAST[25] は、C 言語で書かれたソースコードに対するソフトウェアモデル検査を行うツールである。基本的な考え方は 3節で紹介した SLAM とほとんど同じであり、「2.8 述語抽象化法による検証の手順」で述べた手順に従った検証である。BLAST の大きな特長は、4.3節で詳述する「lazy abstraction」[24] という手法の導入によって、手順に示したループの実行を効率化されることである。

C 言語で使用される構文は、ほとんど全部扱える。主な例外は、関数ポインタと再帰呼び出しである。

Blast 本体は、Object Caml で記述されている。入力である C 言語のソースコードを、C のソースコードを制御フローオートマトン (CFA) に変換しているが、この部分には CIL という、やはり UCB で開発されている C 言語解析のための基盤を使用している。2 つの定理証明器 Simplify と Vampyre が使用されており、前者は抽象構造の遷移関係の計算に、後者は詳細化のための述語の決定に用いられる。

定理証明器の実行時間が全実行時間の大半をしめる。このため、その呼び出しを最小化するための最適化手法が用いられるとともに、定理証明器の 2 回目以降の呼び出しでは、可能ならばキャッシュデータが使用されるようにしている。

表 2: SLAM の実行例

名前	行数	述語数	定理証明器呼出回数	実行時間 (秒)
floppy	6500	23	5509	98
ioctl	1250	5	500	13
openclos	544	5	132	6
srdriver	350	30	3034	93
log	236	6	98	5

出典: [2]

```
    Example() {
 1:   if (*) {
 7:     do {
          got_lock = 0;
 8:       if (*) {
 9:         lock();
            got_lock++;
          }
10:       if (got_lock) {
11:         unlock();
          }
12:     } while (*)
      }
 2:   do {
        lock();
        old = new;
 3:     if (*) {
 4:       unlock();
          new++;
        }
 5:   } while (new != old);
 6:   unlock();
      return;
    }
```

```
lock() {
  if (LOCK == 0) {
    LOCK = 1;
  } else {
    ERROR
  }
}

unlock() {
  if (LOCK == 1) {
    LOCK = 0;
  }else {
    ERROR
  }
}
```

図 9: 例題: ロック

4.3 BLAST での抽象化の方法

BLAST における抽象化の特長は、lazy abstraction という手法である。2.7節で述べたように、述語抽象化法では、使用する述語の数をできるだけ少なくすることが、実行時間の観点から見て重要である。lazy abstraction は、抽象構造の遷移関係の計算を行う場所を分割し、おのおのの場所で使用する述語をその場所で必要最低限のものに制限することによって、同時に使用される述語の数を減らそうというものである。

lazy abstraction アルゴリズムの概要を、図 9を例に説明する。

図 9の左側のコードで、「lock() と unlock() が、必ずこの順で対になって呼ばれる」ことを検証したい。このためには、関数 lock() と unlock() とを図 9の右のように定義して、*ERROR* の部分に到達しないことを検証すればよい[4]。

コードを一見するとわかるように、問題の性質が成立する鍵は、場所によって異なる。ラベル 1 の行が成立した場合の do ループ (ラベル 7 からラベル 12) では、それは `got_lock` という変数であり、そのあとの do ループ (ラベル 2 からラベル 5) では、変数 `old` と変数 `new` との関係である。通常の述語抽象化法では、これら 2 つを扱う述語を用いて全体の抽象構造が構築される。これに対して、lazy abstraction では、以下に見るように、その 2 つの述語をおのおの必要な部分だけで使用することができ、計算量が減らせるのである。

まず、検証対象のソースコードは、図 10に示したような CFA に変換される。各状態は、ソースコード上の同名のラベルに対応している。

次に、初期の述語集合を $\{LOCK = 0\}$ として[5]、「前方エラー探索」フェーズにはいる。このフェーズでは、現在の述語集合で決まる抽象構造の上で CFA をたどって、*ERROR* に達することがあるかどうかを調べる。今の例では、図 11(a) のようになる。仮定より、スタート時点 (状態 1) では、$LOCK = 0$ である。まず CFA の左側 (状態 2) から探索を行う。状態 3 に移るときに lock() を通るので、$LOCK = 1$ となる。次もまず左側の状態 4 を探索する。同様に状態 5 に到達した時点で、$LOCK = 0$ となっている。ここでも左側の状態 6 を探索する。状態 5 から状態 6 への遷移には、ガード [new = old] が書かれているが、現在の述語集合では、この真偽は決定できず、遷移しうるとみなされる。すると、状態 6 の後の unlock() で、*ERROR* に到達する。

[4] 関数 Example() が呼ばれるときには、lock() も unlock() も呼ばれていないものと仮定する。
[5] 初期述語集合は任意である。実際、初期述語集合を空集合として開始しても、同じ結果が得られる。

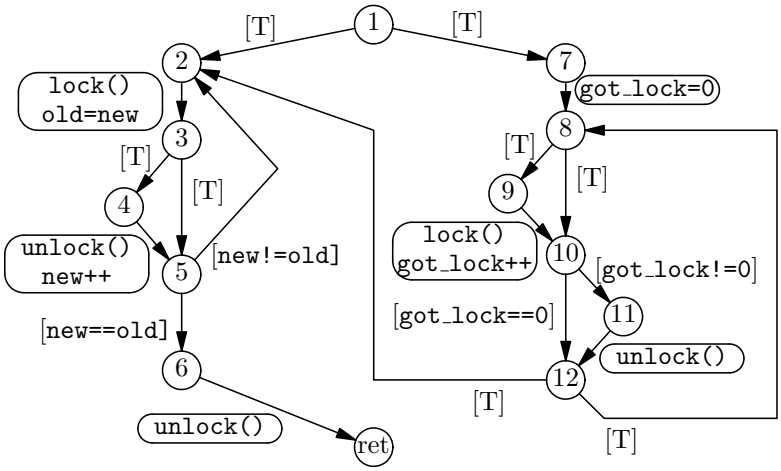
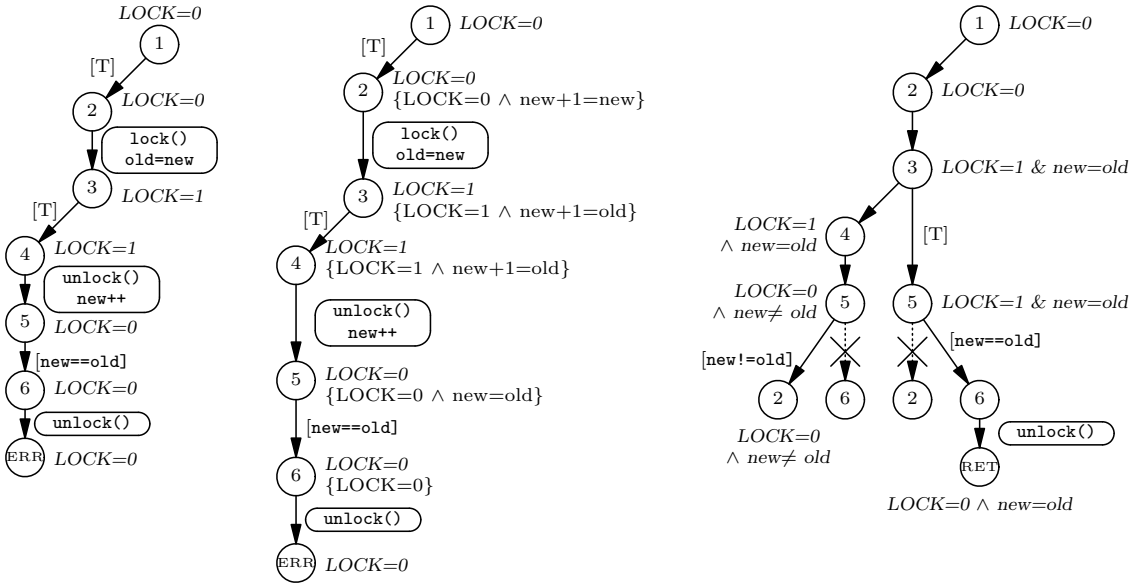


図 10: 制御フローオートマトン



(a) 前方エラー探索　　(b) 後方反例解析　　(c) 詳細化後

図 11: 左側の探索

このようにエラーに到達したら、「後方反例解析」フェーズに移行する。このフェーズでは、エラーに至る実行経路を逆にたどることによって、その実行経路が、もとのプログラムでも実行経路となりうるかどうかを解析する。今の例では図 11(b) のようになる。エラー位置から実行経路を逆にたどり、各状態での最弱前条件を計算していく。この際には、抽象化に用いた述語集合は考慮しない。図では計算された最弱前条件は中括弧の中に示している。たとえば、状態 5 での条件は $LOCK = 0 \wedge new = old$ である。状態 4 から状態 5 への遷移の実行文 `unlock();new++;` に関するこの条件の最弱前条件を求めたものが、状態 4 での条件 $LOCK = 1 \wedge new = old + 1$ である。このような計算を行っていき、最初の状態 1 まで充足可能な条件が得られれば、該当の実行経路は、元のプログラムでも実行経路となりうることがわかる。今の例では、状態 2 において、条件が充足可能ではなくなった。したがって、この実行経路は、元のプログラムには存在しない偽反例であったことがわかる。

偽反例であることがわかった場合には、条件が充足可能ではないことの証明を検討することで、追

加すべき新しい述語が決定できる[6]。この例の場合には、述語 $new = old$ を追加すればよいということがわかる。

そこで、追加された述語集合 $\{LOCK = 0,\ new = old\}$ を用いて、再び「前方エラー探索」フェーズに入るのだが、全体の CFA に対してこれを行うのではなく、前回充足可能でない条件が割り当てられた状態 2 の位置から始める。図 11(c) に示すように、今度は状態 2 からの探索は、すべて、エラーには到達しないで終了する。なお、図中左下の状態 2 は、抽象構造上の $LOCK{=}0\ \&\ new{\neq}old$ によって表されるものである。この条件はすでに到達している (上から 2 段目の) 状態 2 での対応する条件 $LOCK{=}0$ を導くものであるから、ここから新たなエラーは生じないことがわかる。

そこで、バックトラックして探索は状態 1 から右の枝 (状態 7) に移る。述語 $new = old$ を追加したのは状態 2 からの探索についてであるから、ここでは、述語集合はふたたび $\{LOCK = 0\}$ に戻っている。状態 7 からの探索でも、同様にエラーが発見され、後方反例解析の結果、新たな述語 $got\_lock = 0$ が加えられ、述語集合 $\{LOCK = 0,\ got\_lock = 0\}$ での前方エラー探索に移る。この前方探索では、状態 2 に到達した段階で探索を打ち切って良いことに注意する。すでに状態 2 からの解析は終了しているからである。この探索でエラーに到達しないことが確認できる。

このようにして、検証が終了する。全体で使用した述語は $LOCK = 0, new = old, got\_lock = 0$ の 3 つだが、左側の探索でも右側の探索でも述語は 2 つしか使っておらず、同時に使用する述語の数が削減できた。

4.4 適用例

Linux と Microsoft Windows のデバイスドライバに対して BLAST を適用した例が報告されている。結果を表 3に示す。

「述語数」の「全体」は、検証に用いられた述語数の全体、「実効」は、同時に用いられた述語数の最大値である。証明のサイズが空欄になっているのは、誤りが見つかったことを示している。256M

表 3: デバイスドライバへの適用

プログラム	ソース	述語数		全実行時間	反例解析時間	証明のサイズ
	(行数)	全体	実効	(秒)	(秒)	(バイト)
qpmouse.c	23539	2	2	0.50	0.00	175
ide.c	18131	5	5	4.59	0.01	253
aha152x.c	17736	2	2	20.93	0.00	
tlan.c	16506	5	4	428.63	403.33	405
cdaudio.c	17798	85	45	1398.62	540.96	156787
floppy.c	17386	62	37	2086.35	1565.34	
[fixed]		93	44	395.97	17.46	60129
kbfiltr.c	12131	54	40	64.16	5.89	
		48	35	256.92	165.25	
[fixed]		37	34	10.00	0.38	7619
mouclass.c	17372	57	46	54.46	3.34	
parport.c	61781	193	50	1980.09	519.69	102967

出典: [26]

[6] ここではアルゴリズムは省略するが、BLAST が自動的に決定する。

メモリの載った、700MHz Pentium III プロセッサマシンが使用された。上段の 4 つは、Linux kernel のデバイスドライバに対し、lock と unlock の呼び出し方法に関する妥当性検査を行ったもの、下段の 5 つは、Microsoft Windows の DDK (Driver Development Kits) に含まれているデバイスドライバに対して、入出力要求パケットに関する仕様が満足されているかどうかを検証したものである。lazy abstraction によって、使用される述語の数が押さえられていることがわかる。

5 JPF

5.1 ツール要約

名前	JPF (Java Path Finder)
開発元	NASA
主な開発者	W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda
ウェブページ	`http://ase.arc.nasa.gov/visser/jpf/`
ツールの入手	ソースコードがベータテスターと協力者のみに公開される
動作環境	UNIX/MS Windows
特徴	Java のバイトコードを対象とした注釈ベースの検証ツール

5.2 ツール JPF の概要

JPF [7, 22, 35] は注釈ベース (annotation based) の検証ツールであり、 Java のソースコードの中に、 検証したい性質を埋め込んでおく。 埋め込んでおくコードは、 Java のコメントではなく、JPF が提供するクラス `Verfiy` や `Abstract` などのメソッド呼び出しである。図 12 に例を示す。採用されている抽象化は述語抽象化であり、 バックエンドとして決定手続きである SVC (Stanford Validity Checker) [4] を用いる。

(1) 注釈ベース

注釈ベースの検証ツールとは、 ソフトウェアを対象としており、かつ、 検証したい性質を、 検証したいプログラムのソースコード中に直接注釈などの形で書き込むようなツールのことである。利点として、 他の節で紹介する STeP や PAX などのように、 実際に検証したいプログラムに対し、ツールに入力するためのモデルを新たに構築する必要がなく、直接そのソースコードを扱えることがある。また、 ソースコードの中に、 その仕様ともいえる注釈を直接書き込むことによってソースコードの保守がしやすくなるという利点もある。ただし、 SLAM のように、 注釈を記述することなく、 直接ソースコードを扱えるようなツールと比べると、 注釈を記述する分だけ、コストがかかるといえる。 本稿で採り上げる注釈ベースのツールはほかに、ESC/Java がある (7 節)。7.4 節において、ESC/Java における注釈を記述するコストについて議論する。

(2) JPF1 と JPF2

JPF1 [22] では、 検証の対象は Java のソースコードであったが、JPF2 [7] では、 Java のバイトコードになり、 ソースコードを持たない汎用ライブラリを使ったプログラムなども検証対象となった。 また、 JPF1 の出力は、 Spin[38, 27] でモデル検査可能な Promela のソースコードであった

```
class Consumer extends Thread {
  private Buffer buffer;

  public Consumer(Buffer b)
    buffer = b;
    this.start();
  }

  public void run() {
    int count = 0;
    AttrData[] received = new AttrData[10];
    try{
      while (count < 10) {
        received[count] = (AttrData)buffer.get();
        count++;
      }
    }
    catch(HaltException e){};
    Verify.print("Consumer ends");
    Verify.assert("count != 6",count == 6);
    for (int i = 0; i < count; i++){
      Verify.assert("wrong value received",received[i].attr == i);}
  }
}
```

図 12: JPF におけるアサーションの記述の例

が、JPF2 からは、 モデル検査器もツールの一部として含まれており、 JPF2 の出力は、 検証結果である。 検証対象が Java のバイトコードとなったこととあわせて、Promela では扱えない浮動小数点演算を含むプログラムなども検証の対象となった。 本稿では、 JPF2 について紹介する。

(3) 検証の対象

検証の対象は、 任意の Java のバイトコードであり、 マルチスレッドに対応する。 入力は、 検証したい性質が埋め込まれた Java のソースコードである。ただし、 仮定する実行環境は、 通常の JVM (Java Virtual Machine) ではなく、JPF が提供する **MC-JVM** と呼ばれる仮想機械である。 通常の JVM を検証の対象にしようとすると、 簡単に状態数爆発が起こってしまうなど扱いにくいため、MC-JVM は、 検証のためのさまざまな機構をもっている。例えば、 状態数を削減するために、 正規化したヒープ領域の表現 (cannonical heap representation)、 ごみ集め機構 (garbage collection)、構造化された状態表現 (structured state) などの機構をもつ。 さらに、 探索空間の削減のために、 ソースコード中に原子性 (atomicity) を記述できる (図 13)。 また、 Java のプログラムの検証のためには、 外部環境をシミュレートしなければならないが、 そのために MC-JVM は、非決定性をサポートしている。

```
public void initialize(){
  Verify.beginAtomic();
    for (x = 0; x < 10; x++)
      my_array[x] = -1;
  Verify.endAtomic();
}
```

図 13: JPF での原子性の記述

```
Abstract.remove(x);
Abstract.remove(y);
Abstract.addBoolean(“ EQ ”,x==y);
```

図 14: JPF における抽象化の記述

(4) 検証可能な性質

JPF で検証できる性質は、 現在のバージョンでは次の 2 つである。将来的には、 任意の時相論理式で記述される性質も対象とする計画らしい。

1. ソースコード中に埋め込まれた**アサーション** (assertion) が成り立っているか否か。
2. デッドロックに陥るか否か。

アサーションを記述するために、 JPF はクラス `Verify` を提供している。 例えば、 図 12 は、 変数 `count` の値は常に 6 であるというアサーションと、 `receive[i].attr` の値は i であるというアサーションを含む。

5.3 抽象化

JPF は抽象化しなくても、 アサーションが埋め込まれた JAVA のソースコードを、深さ優先探索によってそのまま検証できる。 しかし、 多くの場合、 状態数が多くなりすぎて、 バグの発見など意味のある結果が得られない。 そこで、 抽象化を行うためのクラス `Abstract` が提供されている。クラス `Abstract` で実現される抽象化は、 述語抽象化である。 例えば、 整数変数 `x` と `y` の代わりに、 述語変数 `x==y` を導入したいときは、 図 14のように書く。各プログラムポイントで、 導入された述語変数が真であるか否かは決定手続き SVC で求める。

以上が JPF における述語抽象化についての紹介だが、 前述したとおり、 JPF では、 JVM の代わりに、 探索空間を減らすように工夫された MC-JVM を使っており、 これも抽象化の一つといえる。

5.4 適用例

文献 [7] には、 The Remote Agent(RA) とよばる NASA で開発された、 人工知能ベースの宇宙船制御部のプログラムを、JPF1 の抽象化などを用いて検証した例が紹介されている。また、 文献 [35] には、 DEOS オペレーティングシステムの検証例が紹介されている。 DEOS オペレーティングシステムについては、6.4 節の Bandera の適用例を参照してほしい。

出典:[21]

図 15: Bandera のアーキテクチャ

6 Bandera

6.1 Bandera の要約

名前	Bandera
開発元	Kansas State University
主な開発者	James Corbett, Matt Dwyer, John Hatcliff
web ページ	`http://bandera.projects.cis.ksu.edu/`
ツールの入手	上記 web ページより入手可。執筆時点の最新バージョンは 0.3b2(2003.06.22)
動作環境	UNIX/MS Windows
特長	Java プログラムに対する software model checking を行うためのツール群。

6.2 ツール Bandera の概要

Bandera[14] は、Java で書かれたプログラムに対する検証を行うためのツール群である。アーキテクチャを図 15に示す。特長は、以下のようにまとめられる。

(1) 性質の記述

Bandera は、LTL で書かれた性質の検査を行うように設計されている。しかし、実際のモデル検査の際に、技術者は LTL で直接性質を記述することに困難を感じる場合がある。たとえば、「ウィンドウが開かれてから閉じられるまでの間、ボタン X はたかだか 2 回しか押されない。」という性質を LTL で記述すると次のようになる。

$$\mathbf{G}\,((open \wedge \mathbf{F}\,close) \rightarrow ((\neg pushX \wedge \neg close)\,\mathbf{U}\,(close \vee ((pushX \wedge \neg close)\,\mathbf{U}\,(close \vee ((\neg pushX \wedge \neg close)\,\mathbf{U}\,(close \vee ((pushX \wedge \neg close)\,\mathbf{U}\,(close \vee (\neg pushX\,\mathbf{U}\,close)))))))))$$

このような記述を正しく行うのは容易ではない。Bandera では、この困難さを軽減するために性質の記述言語が用意されている。たとえばこの性質の例だと、次のように記述できる。

```
Between open and close pushX exists atMost 2 times;
```

(2) 性質記述に基づくスライシング

Java のソースコードと検証すべき性質が与えられると、検証する性質に無関係なコードをソースコードから除去するスライシングが行われる。これは、状態の数を減らすと同時に、次の抽象化フェーズで検討すべき変数の個数を抑える効果も持つ。

(3) 抽象構造の構築

Bandera で採用されている抽象構造構築の方法論は、6.3節で詳述する。

(4) バックエンド

以上のステップでスライシングおよび抽象化が行われたモデルは、Jimple[34] で表現されている。これが、一旦、Bandera Intermediate Representation (BIR) と呼ばれる言語での表現に変換される。Bandera 自身は、モデル検査器を持っておらず、BIR で表現されたモデルを、外部のモデル検査器に入力するためのトランスレータを持っている。これにより、Spin, SMV, JPF などのモデル検査器を呼び出すことができる。(ただし、JPF へは BIR を経由せず、直接 Java のコードとして出力される。)

これらのモデル検査器から出力された反例は、ふたたびトランスレータによって BIR に変換され、ここから Jimple に戻されて、エラートレースとしてユーザに報告される。

6.3 Bandera での抽象化の方法

Bandera で採用されている抽象化は、基本的には 2.6節で述べたデータマッピングである。

Bandera においては、抽象化は完全には自動化されておらず、ユーザの指定が必要である。(もちろん、そのために、適切な抽象化を行える可能性が増えているとも言える。) ユーザは、各変数ごとに、マッピングを指定する。

この定義の方法については、次の 2 つが用意されている。

- ビルトインの抽象化方法から選択する。
- ユーザがドメインとマップを定義する。

ちょうど、C 言語において、標準ライブラリ関数も用意されているし、ユーザが自分で関数を定義することもできる、という状況と類似している。ビルトインの抽象化方法には次のものがある。

- **range** 最小値から最大値までの値、および最小値未満、最大値を超えるもの、にマップする。たとえば最小値 3, 最大値 5 とすると、{2 以下, 3, 4, 5, 6 以上 } にマップされる。
- **set** 指定された値のおのおのと、それ以外、にマップする。例えば、値として 3 と 5 を与えると、{3, 5, それ以外 } にマップされる。
- **modulo** 与えられた整数を法とする剰余類にマップする。たとえば 2 を与えると、{$even$, odd} にマップされる。
- **point** 全体を 1 点につぶすようにマップする。

一方、ユーザが抽象化方法を定義する場合には、Bandera Abstraction Specification Language (BASL) と呼ばれる言語を用いる。一例として、整数全体を、$\{NEG, ZERO, POS\}$ にマップする場合の BASL での記述を図 16(a) に示す。(このマップは上述の range の特別な場合であるから、実際にはユーザが定義する必要はない。ここでは BASL の例をあげるためにとりあげた。)

ある変数についての抽象化を指定することにより、別の変数の抽象化の方法の影響を受ける場合

```
abstraction signs abstracts int
begin
  TOKENS = { NEG, ZERO, POS };

  abstract(n)
    begin
        n <  0 -> {NEG};
        n == 0 -> {ZERO};
        n >  0 -> {POS};
    end
  ...
```

(a)BASL による定義

```
public class signs {
  public static final int NEG  = 0; // mask 1
  public static final int ZERO = 1; // mask 2
  public static final int POS  = 2; // mask 4

  public static int abs(int n) {
    if (n <  0) return NEG;
    if (n == 0) return ZERO;
    if (n >  0) return POS;
  }

  public static int add(int arg1, int arg2) {
    if (arg1==NEG  && arg2==NEG)  return NEG;
    if (arg1==NEG  && arg2==ZERO) return NEG;
    if (arg1==ZERO && arg2==NEG)  return NEG;
    if (arg1==ZERO && arg2==ZERO) return NEG;
    if (arg1==ZERO && arg2==POS)  return NEG;
    if (arg1==POS  && arg2==ZERO) return NEG;
    if (arg1==POS  && arg2==POS)  return NEG;
    return Bandera.choose(7);
              /* case (POS,NEG), (NEG,POS) */
  }
  ...
```

(b) 計算された演算

図 16: Bandera での抽象化

がある。たとえば変数 y に変数 x が代入されている場合、変数 y の抽象化は変数 x の抽象化と同じものであることが望ましい。Bandera は、このような依存関係を抽出し、ユーザに報告する。

こうして、(スライシングで生き残った) すべての変数についての抽象ドメインとマップが定義されると、Bandera は、抽象構造を自動生成する。ユーザはマップを定義するだけでよく、遷移関係は、定理証明器 PVS を用いて Bandera が計算する。例として、前述の $\{NEG, ZERO, POS\}$ へのマップについて、加法が計算された結果を図 16(b) に示す。

Bandera での抽象化のもう一つの特長として、「choose-free 探索」がある。2.5節で述べたように、抽象構造におけるモデル検査では、偽反例が報告されうる。したがって、一般には、抽象モデル検査における反例が、具体構造に対応するものがあるかどうかを調べる必要がある。しかし、次の定理が知られている。

定理 6.1 抽象構造における実行経路で、すべての代入が (非決定的でなく) 決定的に行われているものは、具体構造に対応する実行経路がある。 ∎

Bandera ではこれを利用するために、モデル検査器に対し、「非決定的な代入が行われる場合には、直ちにバックトラックする」ように指示をするモードがある。これを「choose-free 探索」と呼んでいる。もちろん、この探索の結果として反例が見つからなかったとしても、抽象構造で (したがって具体構造でも) 反例がないとは限らない。しかし、この探索で反例が見つかれば、必ず具体構造で反例があることが保証される。

6.4 適用例

中程度の規模のマルチスレッド Java プログラムのある程度複雑な性質の検証の例として、Honeywell 社の DEOS オペレーティングシステムへの適用について述べる。

DEOS システムは、マイクロカーネルベースのオペレーティングシステムであり、C++で 1000 行程度のものである。これを、Bandera で検証するために、まず Java のコードへの書き直しが行われた。この OS は、「どのアプリケーションプロセスもバジェット時間内にスケジュールされる 」という性質を満たさない、というバグを持つことが知られており、各種検証ツールでこのバグを発見するかどうかのテストが行われている。

Java への書き直しの結果、20 クラスを持つ 1443 行のコードとなった。メソッド数は 91, フィールド数は 92 である。実行時に、6 つのスレッドが生成される。そのまま検証を行おうとすると状態数が 4GB 以上になるため、何らかの抽象化は必須である。

採用された抽象化は、整数全体をその符号に従って $\{NEG, ZERO, POS\}$ に落とすものである。抽象化を行う変数を発見する手法は、6.3節で述べたものであり、これにより、実行時に無限に増えていく (したがって抽象化を行わなくてはならない) 変数が自動的に発見された。こうして構築された抽象構造に対し、JPF を使用したモデル検査が行われた。

この方法で見つかった (抽象構造上の) 反例は長さ 464 のものである。これが具体構造での反例であるかどうかの決定は困難である。そこで、前述の chooose-free 探索モードでのモデル検査を行い、長さ 318 の反例を得た。これに基づくソースコードの修正後では、抽象モデル上で検証すべき性質が満足されることが示された。

7 ESC/Java

7.1 ツール要約

名前	ESC/Java (Extended Static Checking/Java)
開発元	Compaq
主な開発者	K. Rustan, M. Leino, G. Nelson, J. Saxe
ウェブページ	`http://research.compaq.com/SRC/esc/`
ツールの入手	上記ホームページより入手可。
動作環境	UNIX/MS Windows
特徴	JAVA のソースコードを対象とした注釈ベースの検証ツール

7.2 ツール ESC/Java の概要

ESC/Java [17, 30, 31] は注釈ベースの検証ツールであり、 JAVA のソースコードの中に、 検証したい性質をコメントの形で埋め込んでおく。 また、 名前からもわかるとおり、 プログラムに対し静的な解析だけを行い、 動的な解析は行わない。 ツールの基礎となっている理論は、ダイクストラのガード付きコマンド (Dijkstra's guarded commands) [16] であり、 バックエンドとして定理証明器 SIMPLIFY [15] を使って検証を行う。 モデル検査の技法は使わない。

(1) ESC/Java と ESC/Java 2

ESC/Java の前身は ESC/Modula3 であり、 Compaq Research で開発されてきたが、 現在は ESC/Java 2 となり、 JAVA の仕様記述言語である JML (Java modeling language) [8] のためのツール群のひとつという位置づけになっている。 本稿では、 ESC/Java について紹介する。

(2) とりあげた理由

ESC/Java の大きな特徴として、 健全でも完全でもない検証ツールであることがあげられる。 つまり、 ESC/Java によって検出されたプログラムの「不具合」は、 不具合の可能性だけであって、不具合ではないかもしれないし、 ESC/Java で不具合が 1 つも検出できなかったからといって、 そ

```
 1: public class Vector {
 2:
 3:  Object[] a;
 4:  //@ invariant a != null
 5:  int size;
 6:  //@ invariant size <= a.length
 7:
 8:  public Object elementAt(int i)
 9:  //@ requires 0 <= i && i < size
10:  { ... }
11:
12:  public Object[] copyToArray()
13:  //@ ensures RES != null && RES.length == size
14:  //@ modifies size, a[0], a[*]
     { ... }
   }
```

図 17: ESC/Java の例

のプログラムが正しいとは限らないので、 厳密には検証ツールとはいえない。 ESC/Java の目標は、プログラムの検証というより、 従来の型検査器 (type checker) などでは検出できなかったバグの検出を少ない努力量で行うことである [17]。しかし、 他のソフトウェア検証ツールの多くも、 バグの発見を目的としており、 また ESC/Java もバグの発見には有用であると思われるので、 ここにとりあげた。

(3) 検証の流れ

ユーザはまず、 検証したい JAVA のソースコードを ESC/Java に入力する。ここでは、 いくつかの不具合、 例えば、 null 参照 や 配列の境界越えなどの可能性が指摘される。 ユーザはこれらの指摘にしたがって、 ソースコードを修正してもよい。さらに、 それらについて詳しく検証したいときは、ESC/Java に対する注釈 (annotation) をソースコード中に記述できる。図 17 に例を示す。 ここで、アットマークで始まるコメントが、 ESC/Java に対する注釈であり、 **プラグマ** (pragma) と呼ばれる。 例えば、 図 17 の 4 行目は、 変数 `a` が決して `null` にならないことを主張する。ESC/Java はこれらの注釈にしたがって、 まずダイクストラのガード付きコマンドを生成する [31]。 さらに、 得られたガード付きコマンドから、検証条件 (verification condition) と呼ばれる論理式を生成し、 定理証明器 SIMPLIFY [15] を使って検証条件を証明する。 得られる結果は、正しい (valid) か、 反例か、 または、 証明失敗を意味する発散 (diverges) のいずれかである。

7.3 抽象化

ESC/Java で採用されている抽象化は、述語抽象化である。ESC/Java の前身である ESC/Modula3 では、 特別な機構として抽象化がサポートされていたが、仕様が複雑になるなどの理由により、ESC/Java では、 いくつかのプラグマを組み合わせることにより、 述語抽象化が実現されている。まず、`ghost field` と呼ばれるプラグマが提供されている。 `ghost field` は、 ESC/Java だけが

```
 1:class Bag {
 2:   /*@ non_null */ int[] a;
 3:   int n;
 4:   //@ invariant 0 <= n && n <= a.length;
 5:   //@ ghost public boolean empty;
 6:   //@ invariant empty == (n == 0);
 7:
 8:   //@ requires input != null;
 9:   //@ ensures this.empty == (input.length == 0);
10:   public Bag(int[] input) {
11:     n = input.length;
12:     a = new int[n];
13:     System.arraycopy(input, 0, a, 0, n);
14:     //@ set empty = n == 0;
15:   }
16:
17:   //@ ensures \result == empty;
18:   public boolean isEmpty() {
19:     return n == 0;
20:   }
21:
22:   //@ requires !empty;
23:   //@ modifies empty;
24:   //@ modifies n, a[*];
25:   public int extractMin() {
26:     int m = Integer.MAX_VALUE;
27:     int mindex = 0;
28:     for (i = 0; i < n; i++) {
29:       if (a[i] < m) {
30:         mindex = i;
31:         m = a[i];
32:       }
33:     }
34:     n--;
35:     //@ set empty = n == 0;
36:     //@ assert empty == (n == 0);
37:     a[mindex] = a[n];
38:     return m;
39:   }
40:}
```

図 18: プラグマ ghost field の例

見える変数のようなもので、 JAVA の通常の変数のように扱えるが JAVA のコンパイラからは見えない。 ghost field で宣言された変数の値を更新するためにプラグマ set が用意されている。 プラグマ ghost field およびプラグマ set、 不変式を表すプラグマ invariant などを用いることにより、 述語抽象化を実現できる。 図 18に ghost field を用いた述語抽象化の例を示す。ここで、ghost field の変数 empty は、 具体システムの整数変数 n に対し、$n == 0$ を表すブール変数になっている。

7.4 注釈を記述するコスト

ESC/Java は注釈ベースの検証ツールなので、ソフトウェアの検証を行ううえで、 新たに検証ツールへ入力するモデルを構築するという手間はないが、 3 節で紹介した SLAM などのように、ツールへの入力が、 ソースコードと検証したい性質のみのツールと比べると、注釈を記述しなければならないというコストを考えなければならない。文献 [17] で紹介されている実用例をここで紹介すると、1000 行あたり、 40〜100 行の注釈が必要であり、 一般的なプログラマーで、一時間当たり、 300〜600 行のソースコードに対し、ESC/Java の注釈を記述することができたそうである。

7.5 適用例

文献 [17] には、 ESC/Java の応用例として、Mercator と呼ばれる web 上のプログラムと、 Javafe と呼ばれる JAVA のフロントエンドのプログラムの検証例が紹介されている。Javafe はおよそ 30000 行のプログラムであり、3 週間かけて ESC/Java の注釈を行い、 6 つほどのバグを発見できたそうである。3 週間で 6 つは少ないように見えるが、 プログラムのソースコードの保守を考えると、ESC/Java の注釈は非常に有用であったとのことである。

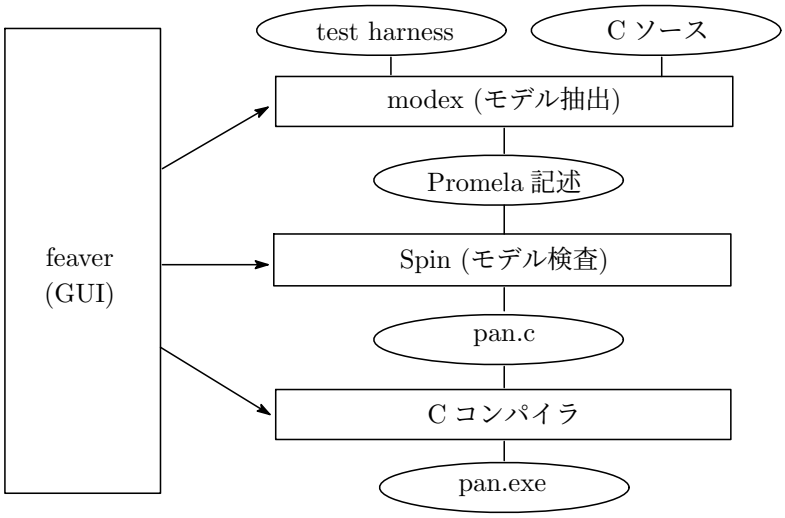


図 19: FeaVer の構成図

8 FeaVer

8.1 FeaVer の要約

名前	FeaVer (Feature Verification)
開発元	ベル研究所
主な開発者	G.J. Holzmann, M.H. Smith
web ページ	`http://cm.bell-labs.com/cm/cs/what/feaver/`
ツールの入手	web ページからダウンロード可能 (v1.0: 2003 年 1 月)
動作環境	Unix (Cygwin も可)
特長	ANSI-C コードの検証ツール。reactive/concurrent なプログラムに対応。

8.2 ツール FeaVer の概要

FeaVer[28] は、ANSI-C コードのソフトウェアモデル検査を支援するツールである。図 19に、構成図を示す。FeaVer には、以下のような特長がある。

(1) 検証する性質の記述

検証したい性質を指定する方法として、FeaVer は以下のものをサポートしている。

- **Promela での記述**　バックエンドのモデル検査器は Spin である。したがって、Spin が認識できる性質の記述方法をとることができる。たとえば以下のものが可能である。
  - assert 文
  - LTL 論理式
  - オートマトン
- **テストダイアグラム**　Promela による性質記述は強力だが、モデル検査になれていない技術者は困難を感じることがある。そのため、「テストダイアグラム」なる指定方法をサポートしている。記述力は限定的だが、直観的にわかりやすく、GUI を用いて入力することができる。

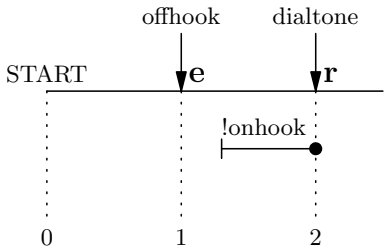


図 20: テストダイアグラム

図 20にテストダイアグラムの例を示す。ここでは、次の内容が表現されている: 受話器をはずす (`offhook`) という「通常イベント (e)」が起こった後、受話器が置かれない (`!onhook`) という「制約条件」の元では、発信音が聞かれる (`dialtone`) という「必須イベント (r)」が起こらなければならない。この例は LTL による記述 $\mathbf{G}(\texttt{offhook} \rightarrow \mathbf{XF}(\texttt{dialtone} \vee \texttt{onhook}))$ と同等である。

(2) モデル抽出

モデル抽出は、MODEX というモジュールが担当する。MODEX は 2 つの入力をとる。1 つは、検証対象となる C ソースコードであり、もう一つは “test harness” と呼ばれるもので、ソースからモデルへの対応ルール (詳細は「8.3 FeaVer での抽象化の方法」を参照)、テスト環境、検証する性質の記述をひとまとめにしたものである。MODEX はこれらからモデルの抽出を行う。抽出されたモデルは、モデル検査器 Spin の入力となるよう、Spin の記述言語 Promela で表現されたものとなる。

(3) モデル検査

FeaVer は、バックエンドモデル検査器として、Spin を使用する。状態空間の探索は、最初はランダムに、次第に精密さを増して実行される。このため、単純で発生しやすいバグ (しばしば重要度が高い) は早く検出され、非常に複雑な状況下でのみ発生するバグ (重要度は低いことが多い) は検出されるまでに時間がかかる。

8.3 FeaVer での抽象化の方法

FeaVer では、抽象化はユーザが定義する。ユーザは、ソースコード中の実行文や (if や while などに現れる) 条件式が、抽出されたモデル中ではどのように対応すれば良いのかを指定していく。つまり、式や文ごとに、ソースからモデルへのマップを与えることになる。典型的なマップには、次のようなものがある。

1. 実行文が、検証すべき性質とまったく関係ない場合。モデルでは、ノーオペレーションにすればよい。生成される Promela のコードは、skip になる。
2. 実行文や条件式が、検証すべき性質と「部分的に」関係する場合。たとえば、タイマーの値の処理をしているとき、値のおのおのには意味がなく、タイマーが切れたかどうか、すなわち値が 0 かどうかだけが意味があるとき。このときには、自然数全体を $\{POS, ZERO\}$ にマップするような変換を行う。

```
extern const int p0;
enum msg_type {Msg, Ack, TimeOut};

void handshake(void)  {
    int resp;

    send(p0, Msg);
    set_timer(16000);  /* msec */

    resp = wait_recv();
    switch (resp) {
    case Ack:
        reset_timer();
        print_log("ack received");
        ...
        break;
    case TimeOut:
        print_log("timeout");
        ...
        break;
    default:
        reset_timer();
        print_log("bad input");
        error("bad input");
        break;
    }
}
```

(a)C ソース

```
active proctype handshake() {
    int resp;

    p0!Msg;
    timer!Set(q0,16000);
    q0?resp;

    do
    :: c_expr {Ack == Phandshake->resp};
        timer!Reset(q0,0);
        skip;
        break; goto C_0
    :: c_expr {TimeOut == Phandshake->resp};
C_0:    skip;
        break; goto C_1
    :: else ->
C_1:    timer!Reset(q0,0);
        skip;
        assert(false);
        break; goto C_2
    od;
C_2:skip;
}
```

(b) 抽出されたモデル

図 21: 例題: ハンドシェーク

変換前	変換後
`set_timer(16000)`	`timer!Set(q0, 16000)`
`reset_timer()`	`timer!Reset(q0,0)`
`send(p0, Msg)`	`p0!Msg`
`resp=wait_recv()`	`q0?resp`
`error(...`	`assert(false)`
`print_log(...`	`skip`

(a) 変換表

```
chan p0 = [0] of {mtype};
chan q0 = [0] of {mtype};

active proctype peer() {
    do
    :: p0?Msg -> if
        :: q0!Ack
        :: q0!Other
        fi
    od
}

active proctype timer_p() {
    chan who = 0;

    do
    :: timer?Set(who,_)
    :: timer?Reset(who,_)
    :: who != 0 -> who!TimeOut
    od
}
```

(b) プロセス定義など

図 22: ソースからモデルへの変換

3. (関数呼び出しなどの) 実行文の詳細な処理内容には意味がないが、とりうる値の範囲には意味がある場合。この場合には、実行文をとりうる値の非決定的な選択に変換する。
4. それ以外の場合。実行文や条件式が、そのままモデル中でも対応する。これが、MODEX ではデフォルトのマップとなる。生成される Promela のコードでは、(Spin v4 で導入された) c_code や c_expr が使用される。

例として、図 21(a) に C のソースを示す。これに対するユーザによる変換定義を図 22(a) に示す。ただし、右側のカラムに現れる Promela の記号などは、図 22(b) のように定義したものである。実際の MODEX では、図 22の内容は、“test harness” として、1 つのファイル中に記述する。

ソース中の実行文 `writeToLog()` は、検証すべき性質に無関係であるので、skip に対応づけられている。タイマーについては、「セットとリセットができ、タイムアウトが発生しうる」という事実だ

けを表すように timer_p という PROMELA のプロセスとして抽象化されている。また、この C ソースで wait_recv() と表されているプロセスの通信相手の処理であるが、ここではその処理内容には興味がない。そこで peer という PROMELA プロセスを定義して、非決定的な選択を行うようになっている。

この C ソースと test harness に対して、MODEX が抽出したモデルを図 21(b) に示す。

以上のように、FeaVer では、抽象化をどのように行うかに関しては、ユーザに大きな自由が与えられている。ユーザの定義によっては、健全でない抽象化になる場合もある。FeaVer による検証では、「プログラムである性質が成立することを証明する」ことを求めるのではなく、「他の手段では発見しにくいバグを (なるべく効率的に) 発見する」ことを追求するべきである。この見地からは、抽象化が健全であるかどうかは、あまり重要ではない。

8.4 適用例

FeaVer システムは、ベル研究所が所属するルーセント・テクノロジー社のパケット交換型アクセスサーバ PathStar のソフトウェアの、呼処理に関する検証を通して開発された。

このソフトウェア (のうちの検証対象部分) は、C で書かれた約 1600 行ほどのプログラムで、可変個のプロセスによって並行して実行される。課金やユーザ管理などの、検証すべき性質とは無関係な関数呼び出しは、変換表によってふるい落とされた。

コード全体の実行文のうち、約 60%は検証すべき性質に関係するコードであり、抽象化されずにそのままモデルとして抽出された。完全に無関係であって、ノーオペレーション (skip) に落とされた実行文はおよそ 30%である。残りの 10%は部分的に関係するコードであり、個別にマッピングが定義された。

PathStar の呼処理が持つ 20 あまりの機能に対して、検証すべき性質は合計 80 程度であった。複数の機能が同時に起こりうるかどうか、などの制約条件を考慮した結果、約 200 程度の検証の実行が必要となった。

検証は、ソフトウェアの設計/構築/保守フェーズを通して 18ヶ月の期間行われた。この間、日々更新されるソフトウェアのバージョンは 300 を超え、最初期のバージョンに比べ、最後期のバージョンのコード量は 5 倍程度に増えている。検証は繰り返し行われ、およそ 75 の誤りが発見され、多くは重要な誤りであった。検証チームとは独立に設けられていた、通常の方法によるシステムテストチームでは、これらのうち 5 個を発見するにとどまっている。

FeaVer ツールは、(通常の方法による) テスト時の補助ツールとしても有用だったことが報告されている。並行プロセスに対するテストであるから、起こったエラーの再現が困難な場合が多い。そのようなとき、テストでの各イベントの発生順序を FeaVer に与えることによって、どの競合条件がエラーを引き起こすかを調べることができた。

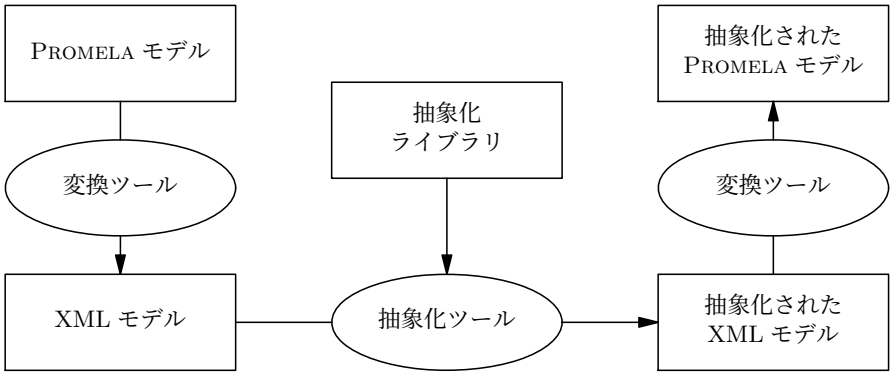


図 23: αSPIN 概要図

9 αSPIN

9.1 αSPIN の要約

名前	αSPIN。α は抽象化関数の記号からとった
開発元	Malaga 大学 (スペイン)
主な開発者	M.M. Gallardo, J.M. Cruz, P.M. Gomez, E. Pimentel
web ページ	`http://polaris.lcc.uma.es/~gisum/fmse/tools/mainframe.html`
ツールの入手	Web ページから入手可 (2003/01 v0.9)。
動作環境	MS Windows
特長	PROMELA で記述されたモデルを抽象化するツール。抽象化の表現には XML を用いる。

9.2 ツール αSPIN の概要

αSPIN[18] は、モデル検査ツール SPIN を対象にした抽象化ツールである。SPIN の記述言語である PROMELA で記述されたソースを入力として受け取り、それに抽象化を施し、再び PROMELA で記述したソースを出力する。想定される使用方法は、PROMELA でモデルを記述して SPIN でモデル検査を行ったとき、状態数が多すぎるために検査が現実的な時間で終了しない場合に、αSPIN を用いて抽象化を行う、というものである。

抽象化は、2.6節で述べたデータマッピングによるものであり、マッピングはユーザが定義するほか、ライブラリとして登録されている他のユーザが作成したマッピングを使用することもできる。詳細については「9.3 αSPIN での抽象化の方法」を参照。

現在の αSPIN は、PROMELA が対象言語であるが、将来、別の言語への拡張を可能にするため、次のような工夫がなされている (図 23を参照)。

- αSPIN の入力 (具体構造) と出力 (抽象構造) は、PROMELA とは独立な構造記述言語で与える。この記述言語は、XML ベースのものである。
- PROMELA による構造記述と、XML ベースの構造記述を互いに変換する機能が用意されており、SPIN の GUI の中から呼び出せるようになっている。

PROMELA による記述と、対応する XML ベースの記述の例を、図 24に示す。

```
active proctype  AC(){

 do
 :: c ? UP ->
        if
        ::  T < H -> T++
        :: else -> skip
                fi

 :: c ? DOWN ->
        if
                :: T > L -> T--
        :: else -> skip
        fi
 :: c ? TO20 -> T = 20
 od

}
```

```
<proc  instances="1"  type="PROCTYPE"  name="AC">
  <body>
    <do>
      <option>
        <receive rtype="normal">
          <channel><var name="c"></var></channel>
          <receive_arguments><const>UP</const></receive_arguments>
        </receive>
        <if>
          <option>
            <expression type="LT">
              <left><var name="T"></var></left>
              <right><const>30</const></right>
            </expression>
            <expression type="INCR">
              <left><var name="T"></var></left>
            </expression>
          </option>
          <option>
            <else/>
            <const>1</const>
          </option>
        </if>
      </option>
      <option>
      ...(略)...
    </do>
  </body>
</proc>
```

図 24: Promela の記述と対応する XML 記述

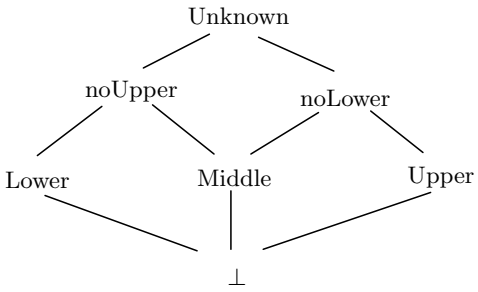


図 25: 階を抽象した束構造

9.3 αSPIN での抽象化の方法

αSPIN でのデータマッピングによる抽象化の方式を、エレベータ制御の例で説明する。

エレベータが移動する階の範囲を $[1..N]$ とする。これを抽象化したものとして、図25に示す束構造を考える。抽象化関数 β は、$\beta(1) = \text{Lower}$, $\beta(N) = \text{Upper}$, $\beta(x) = \text{Middle}\ (1 < x < N)$ で与えられる。対応するガロア接続を (α, γ) とすれば、$\gamma(\text{noUpper}) = \{1, \ldots, N-1\}$, $\alpha(\{N-1, N\}) = \text{noLower}$ などとなる。

さて、Promela 記述の基本部分は、評価式および実行文から成り立っている。たとえば、i がエレベータの現在の階を表す変数だとして、次のコード

```
if
:: i == 0  ->  i = i + 1;
fi
```

において、`i == 0` は評価式、`i = i + 1` は実行文である。αSPIN では、評価式および実行文を変換していくことによって、抽象構造を表す Promela 記述を生成する。

評価式は、$t : D \to \{true, false\}$ なる関数を表現しているとみることができる。これに対し、$\overline{t} : \overline{D} \to \{true, false\}$ なる関数を $\overline{t}(\overline{d}) = \bigvee\{\ t(d) \mid \beta(d) = \overline{d}\ \}$ で定義する (表 4の右側)。これを表現する評価式が、もとの評価式の変換結果となる。実行文についても同様である。評価式 `i == 0` と実行文 `i = i + 1` について、この考え方によって、上のコードは次のように変換される。

```
#define FLR_EQ(x,y) \
   ( (x==y) || ((x==Upper)&&(y==NoLower)) || ((x==NoLower)&&(y==Upper)) \
    || ((x==Lower)&&(y==NoUpper)) || ((x==NoUpper)&&(y==Lower)) || ((x==Middle)&&(y==NoUpper)) \
    || ((x==NoUpper)&&(y==Middle)) || ((x==Middle)&&(y==NoLower)) || ((x==NoLower)&&(y==Middle)) \
    || (x==Unknown) || (y==Unknown) )

inline FLR_INCR(v) {
  if
  :: (v == Lower)                                -> v = Middle;
  :: (v == Middle || v == NoUpper || v == NoLower) -> v = NoLower;
  :: (v == Upper || v == Illegal)                -> v = Illegal; assert(0);
  fi
}
```

図 26: Promela コードの変換

出典: [18]

図 27: エレベータ制御における探索空間のサイズ

```
if
:: FLR_EQ(i,0)  ->  FLR_INCR(i);
fi
```

ただし、FLR_EQ や FLR_INCR は、図 26で定義されるものである。

現在の αSPIN のバージョンでは、これらの変換は、ユーザが与えなければならない[7]。PVS などを用いて、変換を自動的に生成することが開発チームによって検討されている。

表 4: 実行文と評価式の抽象構造での解釈

i の値	実行文 `i = i + 1` の結果	評価式 `i == 0` の値
Lower	Middle	true
Middle	noLower	false
Upper	$\perp$	false
noLower	noLower	false
noUpper	noLower	true
Unknown	Unknown	true

9.4 適用例

エレベータ制御における性質の検証に、前述の抽象化を用いた例が報告されている。階数を変えたときの探索空間の大きさを図 27に示す。図中の Deadlock はシステムがデッドロックを持たないことの検証である。Move は LTL 論理式 **G**(((reqL ∧ posU) →**F**(posBelowU)) ∧ ((reqU ∧ posL) →**F**(posAboveL)) ∧ ((reqM ∧ noPosM)→**F**(posM))) の検証である。ここで、X が L(ower), M(iddle) などの場所を表すとして、req$X$ は、「X でエレベータが呼ばれた」ことを、pos$X$ は、「X にエレベータがある」ことを表す。つまり、これは「エレベータは、いつかは呼ばれた階に到着する」ことの検証である。

図から、具体モデルでは階数とともに状態数が爆発的に増大すること、それに対して抽象モデルでは、状態数増加が線形に抑えられていることが読みとれる。

10 STeP

10.1 ツール要約

名前	STeP (The Stanford Temporal Prover)
開発元	Stanford University
主な開発者	Z. Manna
ウェブページ	`http://www-step.stanford.edu/`
ツールの入手	上記ホームページより入手可。 最新バージョンは 1.3。
動作環境	UNIX
特徴	実時間システムを含む刺激応答システムの LTL 式による性質の検証ツール

10.2 ツール STeP の概要

STeP [13] は実時間システムを含む刺激応答システムを表す公平遷移システムで表された検証の対象に対し、 LTL 式で表された性質を検証するためのツールである。 無限状態システムを検証対象としていることが特徴である。検証の流れは、 以下のようにまとめられる。

(1) 検証の対象の記述

STeP における検証対象は、 **公平遷移系** (fair transition system, FTS) [29] で与える。 公平遷移系とは、 2.2 節で紹介した遷移系の特殊な場合であり、 $(\mathcal{V}, \Theta, \mathcal{T})$ の 3 つ組で与えれらる。 ここで、$\mathcal{V}$ は **変数** (variable) の集合であり、 公平遷移系の状態集合は、 変数への値割り当て全体である。 変数の領域として無限集合を考えることができるので、 一般には無限状態システムである。 変数上の一階の式を**アサーション** (assertion) と呼ぶ。 JPF や ESC/Java におけるアサーションとは違うので注意してほしい。 アサーション ϕ と状態 s に対し、 s が ϕ を満たすとき、 $s \models \phi$ と書くとすると、アサーション ϕ に対し、 状態の集合 $\{s \text{ は状態} \mid s \models \phi\}$ を考えるとアサーションは状態の集合を表すと考えることができる。 Θ は**初期条件**(initial condition) と呼ばれるアサーションであり、初期状態の集合を表す。 変数の集合 $\mathcal{V}$ に対し、$\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$ を **点付き変数**(primed variable) の集合とし、 $\mathcal{V} \cup \mathcal{V}'$ 上のアサーションを $\rho(\mathcal{V}, \mathcal{V}')$ と書く。 $\mathcal{T}$ は **遷移**(transition) の集合で、 遷移は $\mathcal{V} \cup \mathcal{V}'$ 上のアサーションであり、 次のように遷移関係を定義する。 状態 s_0, s_1 に対し、 ある遷

[7] 変換定義を集めたライブラリを使うこともできる。

$$\textbf{local } x\colon \{0,1,2\} \textbf{ where } x = 0$$

$$P_1 :: \left[\begin{array}{l} l_0\colon \textbf{await } x = 0 \\ l_1\colon x := 1 \\ l_2\colon \textbf{skip} \\ l_3\colon \textbf{await } x = 1 \\ l_4\colon \textbf{critical} \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} m_0\colon \textbf{await } x = 0 \\ m_1\colon x := 2 \\ m_2\colon \textbf{skip} \\ m_3\colon \textbf{await } x = 2 \\ m_4\colon \textbf{critical} \end{array}\right]$$

図 28: Fischer の排他制御アルゴリズム

移 $\rho(\mathcal{V}, \mathcal{V}') \in \mathcal{T}$ が存在し、 $\rho(s_0, s_1)$ が真になるとき、 s_0 と s_1 が遷移関係を持つとする。 このように、 点付き変数は、 遷移関係において、 「次」の状態の条件を表している。また、 それぞれの遷移に対し、 **弱い公平性** (weak fairness, just) および**強い公平性** (strong fairness, compassionate) を仮定できる。弱い公平性をもつ遷移 ρ_w は、 ρ_w が続けて無限回遷移可能なら、ρ_w によって必ず遷移し、 強い公平性をもつ遷移 ρ_s は、ρ_s が無限回遷移可能なら、 ρ_s によってかならず遷移する。公平遷移系 $(\mathcal{V}, \Theta, \mathcal{T})$ の状態全体を Σ と書き、3 つ組 $(\Sigma, \Theta, \mathcal{T})$ で与えることがある。公平遷移系 $\mathcal{S} = (\mathcal{V}, \Theta, \mathcal{T})$ の**計算**(computation) とは、状態の無限列 $s_0, s_1, \ldots$ で、 次の条件を満たすものと定義される。

1. s_0 は初期条件を満たす ($s_0 \models \Theta$)。
2. 任意の $i \geq 0$ について、 ある $\rho \in \mathcal{T}$ によって $\rho(s_i, s_{i+1})$ を満たす。
3. 公平性の仮定を満たす。

変数の 1 つに、 実数変数 `Clock` を仮定し、 無条件で `Clock` が一定の値増える遷移を仮定すれば、実時間システムを表現することができる。 `Clock` のような時刻を表す変数 (複数あってもよい) をもつ公平遷移系を時計付き遷移系 (clocked transition system, CTS) と呼ぶことがある。

例 10.1 ここで、公平遷移系の例として、Fischer の実時間排他制御アルゴリズム (Fischer's real-time mutual exclusion algorithm) をとりあげる。文献 [29] の表記法によるアルゴリズムを図 28 に示す。簡単に説明すると、2 つのプロセス P_1 と P_2 があり、$\|$ は、 非同期並列結合 (asynchronous parallel composition) を表す。 **await** 命令は、 その条件が成り立つまでそこで待つことを意味する。 各遷移可能な遷移は、 L 時間以上 U 時間以下待つこととしたとき (L と U は実数)、 プロセス P_1 の l_4 行目 と プロセス P_2 の m_4 行目とが同時に実行しないことを検証するのが目的である。 次に、 Fischer の実時間排他制御アルゴリズムに対応する公平遷移系を定義する。 まず、 変数の集合は $\mathcal{V} = \{x, \pi_1, \pi_2, c_1, c_2\}$ であり、 それぞれ、 x はプログラム中の x、 π_i はプロセス P_i のプログラムポイント、 実数変数 c_i はプロセス P_i の局所時刻を表すとする (ここで、 i は 1 か 2)。初期条件 Θ は

$$\pi_1 = l_0 \wedge \pi_2 = m_0 \wedge c_1 = 0 \wedge c_2 = 0 \wedge x = 0$$

となる。 遷移は、 例えば、 l_1 に対応する遷移 ρ_{l_1} は

$$\rho_{l_1}\colon \left(\begin{array}{c} \pi_1 = l_1 \wedge c \geq L \wedge \\ \pi_1' = l_2 \wedge c_1' = 0 \wedge x' = 1 \end{array}\right) \wedge \pi_2' = \pi_2 \wedge c_2' = c_2$$

となる。 ∎

(2) 性質の記述

STeP における性質は、 LTL 論理式で与えられる。 原子述語は、 $x = 0$ など、 変数に関する命題である。 状態の無限列 σ と LTL 論理式 f に対し、σ の i 番目の状態で f が真であるとき、$(\sigma, i) \models f$ と書く。 例として、前述した Fischer の排他制御アルゴリズムの安全性は、 LTL 論理式で

$$\mathbf{G}\neg(\pi_1 = l_4 \wedge \pi_2 = m_4)$$

と書ける。

(3) モデル検査

STeP は構成要素として、 モデル検査器をもつ。与えられた検証対象が有限システムであった場合、 つまり、 すべての変数の領域が有限であった場合、 STeP は与えられた有限システムと性質に対し、 モデル検査による検証を行う。 また、 無限システムに対しても、 モデル検査により、 バグの検出を行うこともできる。

(4) 証明支援系

STeP は構成要素として、 証明支援系をもつ。 与えられた検証対象が無限システムであった場合、STeP は証明支援系による検証を行うことが出来る。 証明支援系は不変式を自動生成できる。 得られた不変式は、 以下で述べる抽象化の過程で用いることができる。

(5) 抽象化

STeP は構成要素として、 抽象化ツールをもつ。 与えられた検証対象が無限システムであった場合、 STeP は、 その検証対象の有限状態の抽象システムを求めるための支援をすることができる。得られた有限状態の抽象システムに対しモデル検査を行うことによって、 具体システムを検証する。抽象化の方法および抽象化の正当性については、 次の節で詳しく述べる。

10.3 抽象化

2 つの公平遷移系 $\mathcal{S}$ と $\mathcal{A}$ を考える。 $\mathcal{S}$ の状態空間を Σ と書くと、 $\mathcal{S}$ におけるアサーション全体は、Σ の部分集合全体を表しているとみなせるので、 2^Σ と書ける。$\mathcal{A}$ の状態空間が順序をもつ集合 $(\Sigma^\mathcal{A}, \preceq)$ で、$(2^\Sigma, \subseteq)$ と $(\Sigma^\mathcal{A}, \preceq)$ が、 関数 $\alpha\colon 2^\Sigma \to \Sigma^\mathcal{A}$ と $\gamma\colon \Sigma^\mathcal{A} \to 2^\Sigma$ によってガロア接続を持つとき、γ は次のようにして、$\mathcal{A}$ でのアサーション全体 $2^{\Sigma^\mathcal{A}}$ および $\mathcal{A}$ の遷移関係へ拡張できる。 すなわち、 アサーション $S \in 2^{\Sigma^\mathcal{A}}$ および遷移 $\rho^\mathcal{A}\colon \Sigma^\mathcal{A} \times \Sigma^\mathcal{A}$ に対し、 それぞれ、$\gamma(S) = \bigcup_{a\in S} \gamma(a)$ および $\gamma(\rho^\mathcal{A}) = \{\langle s_1, s_2\rangle \mid s_2 \in \gamma(a_2) \text{ and } s_2 \in \gamma(a_2) \text{ for some } \langle a_1, a_2\rangle \in \rho^\mathcal{A}\}$ と定義する。 以上の拡張に基づいて、 述語抽象化の一種である**アサーションに基づく抽象化** (assertion-based abstraction) を定義する。 まず、 アサーションの有限集合 B を与える。B およびブール演算子 ($\wedge, \vee$ および $\neg$) から作られる式全体 ($\mathcal{BA}(B)$ と書く)、 およびそれらの間の含意関係 ($\preceq$ と書く) を、 **基礎 B によるアサーションに基づく抽象領域**(assertion-based abstract domain with basis B) と呼ぶ。 具体化関数 γ および 抽象化関数 α は、 それぞれ $\gamma(f) = \{s \in \Sigma \mid s \models f\}$ および $\alpha(S) = \bigwedge\{s \in \mathcal{BA}(B) \mid S \subseteq \gamma(s)\}$ と定義される。 2.3 節の 定理 2.3 に対応する公平遷移系に対する定理は次のようになる。

定理 10.2 [13] 公平遷移系 $\mathcal{S} = (\Sigma, \Theta, \mathcal{T})$、アサーションの有限集合 B に対し、$\mathcal{A} = (\mathcal{BA}(B), \Theta^{\mathcal{A}}, \mathcal{T}^{\mathcal{A}})$ を次の条件を満たす公平遷移系とする。

1. 初期条件に関して、 $\Theta \subseteq \gamma(\Theta^{\mathcal{A}})$ を満たす。
2. $\mathcal{S}$ の任意の遷移 $\rho \in \mathcal{T}$ に対し、 $\mathcal{A}$ の遷移 $\rho^{\mathcal{A}} \in \mathcal{T}^{\mathcal{A}}$ が存在し、$\rho \subseteq \gamma(\rho^{\mathcal{A}})$ を満たす。
3. $\mathcal{A}$ の遷移 $\rho^{\mathcal{A}} \in \mathcal{T}^{\mathcal{A}}$ が弱い (強い) 公平性をもつとき、 次を満たす弱い公平性をもつ遷移 $\rho \in \mathcal{T}$ が存在する。
   (a) $\gamma(\mathit{enabled}(\rho^{\mathcal{A}})) \subseteq (=)\mathit{enabled}(\rho)$
   (b) $\mathit{post}(\rho, \gamma(\mathit{enabled}(\rho^{\mathcal{A}}))) \subseteq (=)\gamma(\mathit{post}(\rho^{\mathcal{A}}, \mathit{enabled}(\rho^{\mathcal{A}})))$
   ここで、 $\mathit{enabled}(\rho)$ および $\mathit{post}(\rho, \phi)$ は、それぞれアサーション $\exists\mathcal{V}'.\rho(\mathcal{V}, \mathcal{V}')$ および $\exists\mathcal{V}_0.(\rho(\mathcal{V}_0, \mathcal{V}) \wedge \phi(\mathcal{V}_0))$ の略である。

このとき、 $\mathcal{A}$ は $\mathcal{S}$ に対し、 ACTL* 論理式に関して健全な抽象化である。 ∎

上の定理の条件 1 および 2 は、 2.3 節の 定理 2.3 の条件と同じである。条件 3 は、 公平性に関する条件である。 STeP は、 与えられた基礎 B に基づいて、 これらの条件を満たす抽象システムを自動的に生成する。

例 10.3 例 10.1 でとりあげた Fischer の排他制御アルゴリズムを表す公平遷移系に対する基礎として、 次のような集合を考える。

$$\begin{array}{ll} b_1 \colon c_1 \geq L & b_4 \colon c_2 \geq c_1 \\ b_2 \colon c_2 \geq L & b_5 \colon c_1 \geq c_2 + L \\ b_3 \colon c_1 \geq c_2 & b_6 \colon c_2 \geq c_1 + L \end{array}$$

このとき STeP は、 定理 10.2 の条件を満たす公平遷移系を自動的に生成する。 初期条件は、

$$\pi_1 = l_0 \wedge \pi_2 = m_0 \wedge x = 0 \wedge \neg b_1 \wedge \neg b_2 \wedge b_3 \wedge b_4 \wedge \neg b_5 \wedge \neg b_6$$

となる。 遷移は、 例えば、 l_1 に対応する ρ_{l_1} は、

$$\rho^{\mathcal{A}}_{l_1} \colon \left(\begin{array}{c} \pi_1 = l_1 \wedge \pi_1' = l_2 \wedge x' = 1 \wedge \pi_2' = \pi_2 \wedge \\ b_1 \wedge \neg b_1' \wedge b_4' \wedge \neg b_5' \wedge (b_2 \rightarrow \neg b_3') \wedge (b_2 \rightarrow b_6') \wedge \\ (\neg b_3 \rightarrow \neg b_3') \wedge (\neg b_3 \rightarrow b_6') \wedge (b_4 \rightarrow \neg b_3') \wedge (b_4 \rightarrow b_6') \wedge \\ (\neg b_5 \rightarrow \neg b_3') \wedge (b_6 \rightarrow \neg b_3') \wedge (b_6 \rightarrow b_6') \end{array} \right)$$

と抽象化される。 ∎

10.4 適用例

抽象化を用いない STeP の適用例としては、 リーダ選択問題 (leader election problem) や一般化踏切問題 (generalized railroad crossing problem) [6] など の検証が報告されているが、 抽象化を用いる例としては、 2 つの排他制御アルゴリズムと 2 つの通信プロトコルの検証例 [13] が報告されている。 いずれも、 無限状態システムである。4 つの例の名前、 遷移の数、 アサーションに基づく抽象化に用いられた基礎の数、抽象化および得られた抽象システムに対するモデル検査に要した時間をまとめた表を表 5 に示す。

表 5: STeP の適用例における実行時間

System	transitions	basis size	abstraction time	model check time
Bakery	14	3	3s	<1s
Fischer	11	6	28s	1s
Alternating-bit	7	4	14s	<1s
Bounded Retransmission	13	7	70s	4s

出典: [13]

11 PAX

11.1 PAX の要約

名前	PAX (Parameterized systems Abstracted and eXplored)
開発元	Christian-Albrechts-Universität zu Kiel (ドイツのキール大学)
主な開発者	K. Baukus, Y. Lakhnech, K. Stahl
ウェブページ	`http://www.informatik.uni-kiel.de/~kba/pax/`
ツールの入手	不可
特徴	パラメータ化ネットワークを検証するための抽象化ツール

11.2 ツール PAX の概要

PAX[5] は、 無限状態遷移系の特殊な場合であるパラメータ化ネットワークを検証するための抽象化ツールである。 採用されている抽象化は、 前節の STeP で紹介したアサーションに基づく抽象化であるが、 基礎はユーザが与えるのではなく、 パラメータ化ネットワークに依存した形の基礎を用い、 抽象化は全自動で行われる。 得られる抽象システムは有限状態遷移系であり、SMV や Spin でモデル検査することによって検証を行う。 その特徴および検証の流れは以下のようにまとめられる。

(1) 検証の対象

PAX の検証対象は、 **パラメータ化ネットワーク**(parametrized network) [5] である。 パラメータ化ネットワークとは、 遷移系の集合 $\{\mathcal{P}_i \mid i \in \omega\}$ のことで、 $\mathcal{P}_i$ は、 i 個の有限状態遷移系の非同期並列結合 (asynchronous parallel composition) [29] を表す。 例えば、 任意の数のプロセスに対して記述された排他制御アルゴリズムなどが検証の対象である。 パラメータ化ネットワークは、 無限状態遷移系の特殊な場合とみなせるが、 一般のパラメータ化ネットワークに対する検証問題は決定不能である [1]。

PAX で扱うパラメータ化ネットワークは、 WS1S (weak second order theory of one successor) [9, 33] と呼ばれる 2 階論理の理論の論理式によって定義される。 WS1S の項 (term) は、 定数 0 と 1 階の変数と関数記号 *succ* から構成されるもので、 自然数に対応する。 原子論理式 (atomic formula) は、

$$b, \quad t = t', \quad t < t', \quad t \in X$$

である。 ここで、 b はブール値をとる変数、 t と t' は項、 X は 2 階の変数である。 WS1S 論理式とは、 原子論理式および論理結合子 $\neg, \vee$、1 階および 2 階変数への限量子 $\forall, \exists$ からなるもので、1 階の変数は自然数へ、 2 階の変数は自然数の有限集合へ解釈される。

PAX で扱うパラメータ化ネットワークは、 次のような、 **公平単項的遷移システム** (fair monadic transition system) によって与えられる。 公平単項的遷移システムは、 同じような形の任意の個数の有限状態遷移系をもつ公平遷移系を与えるための、 一種のスケルトンまたはテンプレートのようなものであり、 2 つの自然数のパラメータ i と n を決めると、 1 つの公平遷移系が決まるのもので、 $\mathcal{S}(i,n)$ と書き、 次の 3 つ組み $(\mathcal{V},\Theta,\mathcal{T})$ で与えられる。 $\mathcal{V}$ は、 長さ n のブール値の配列を表す変数の集合であり、 ブール配列変数 (boolean array variable) と呼ばれる。Θ は初期条件を、 $\mathcal{T}$ は遷移関係を表し、 自然数のパラメータ n によって決まる WS1S の部分クラスである AF(n) と呼ばれる論理の式である [5]。 それぞれの遷移に対して、 弱い公平性および強い公平性を仮定することができる。 また、 Θ と ρ に現れる自由変数は、 i を変数とみなすとそれぞれ、 $\mathcal{V}\cup\{i\}$ および $\mathcal{V}\cup\mathcal{V}'\cup\{i\}$ である。 公平単項的遷移システム $\mathcal{S}(i,n)$ は、 2 つのパラメータ i と n を与えることによって、 変数 i のところだけが違う同じような形をした n 個の公平遷移系を生成することができる。公平遷移系の 非同期並列結合関係を $\|$ と書くと、 m 個の $\mathcal{S}(i,n)$ で生成された公平遷移系の非同期並列結合

$$\mathcal{P}_m \;=\|_{0\le l\le m-1}\,\mathcal{S}(l,m) \tag{18}$$

を考えることができる。 $\mathcal{P}_m$ は公平遷移系なので、 遷移も通常通り定義される。 検証の対象は、 公平単項的遷移システム $\mathcal{S}(i,n)$ を与えたときに (18) 式で得られる公平遷移系の非同期並列結合のすべて

$$\mathcal{P}=\{\mathcal{P}_m \mid m\ge 1\} \tag{19}$$

であり、 **単項的パラメータ化システム** (monadic parameterized system, MPS) と呼び、 パラメータ化ネットワークである。 MPS そのものは、公平遷移系の集合であって、 公平遷移系ではないことに注意したい。 上記のとおり、 公平単項的遷移システムによって、 MPS が与えられる。

例 11.1 Szymanski の排他制御アルゴリズム (Szymanski's mutual exclusion algorithm) [32] の $\mathcal{S}(i,n)$ による記述を図 29に示す。任意のプロセス i と j について、 クリティカルセクションを表す ℓ_7 行目に同時に入らないように考えられたアルゴリズムであり、直感的な説明は文献 [32] を参照してほしい。アルゴリズム中の **await** 命令 (await statement) は、 遷移の条件を表す。 つまり、 **await** 以下の条件が成り立つまで、 その場所で待つ [29]。 例えば、 ℓ_1 から ℓ_2 への遷移に対応する AF(n) の式による遷移は次のようになる。

$$\begin{aligned}&(\forall_n j\colon \mathrm{at}\_\ell_1[j]\vee\mathrm{at}\_\ell_2[j]\vee\mathrm{at}\_\ell_4[j])\wedge\forall_n j\colon j\neq i\to\textstyle\bigwedge_{1\le l\le 7}\mathrm{at}\_\ell_l[j]\leftrightarrow\mathrm{at}\_\ell'_l[j]\\&\quad\wedge\mathrm{at}\_\ell_1[i]\wedge\neg\mathrm{at}\_\ell'_1[i]\wedge\mathrm{at}\_\ell'_2[i]\wedge\textstyle\bigwedge_{3\le l\le 7}\mathrm{at}\_\ell_l[i]\leftrightarrow\mathrm{at}\_\ell'_l[i]\end{aligned}$$

ここで、 $\forall_n j\colon f$ は、 $\forall j\colon j\le n\to f$ の略記である。 ∎

次に、 **WS1S 遷移系** (WS1S transition system) を定義する。 MPS は遷移系の集合であったが、WS1S 遷移系は遷移系であり、 与えられた MPS から、等価な WS1S 遷移系へ変換できる。 WS1S 遷移系は 3 つ組み $(\mathcal{V},\Theta,\mathcal{T})$ で与えられ、 ここで、 $\mathcal{V}$ は有限個の 2 階変数で、 Θ は初期条件を表す $\mathcal{V}$ 上の WS1S 論理式であり、 $\mathcal{T}$ は、 遷移を表す $\mathcal{V}\cup\mathcal{V}'$ 上の WS1S 論理式の集合である。それぞれの遷移に対して、 弱い公平性および強い公平性を仮定できる。 MPS から WS1S 遷移系へのへの変換を次に述べる。 MPS $\mathcal{P}$ が m 個のブール配列変数をもつ $\mathcal{S}(i,n)$ で与えられるとする。 $\mathcal{P}$ の要素の遷移系 $\mathcal{P}_l=P_0\parallel\cdots\parallel P_{l-1}$ の状態は、m 個の長さ l のブール値のベクトルで表現できる (図 30)。ここで、 ベクトル $(s_0,\ldots,s_{l-1})$ に対し、 自然数の集合 $\{i\mid s_i \text{ は真}, 0\le i\le l-1\}$ を対応させれば、 $\mathcal{P}_l$ の状態は m 個の 2 階の変数で表現できる。 ほかに、 n 個のプロセスまでのインデックスを表現する 2 階変数 P を用意しておく。

ℓ_1: **await** $\forall_n j : \mathrm{at}\_\ell_1[j] \vee \mathrm{at}\_\ell_2[j] \vee \mathrm{at}\_\ell_4[j]$
ℓ_2: **skip**
ℓ_3: **if** $\exists_n j : \mathrm{at}\_\ell_2[j] \vee \mathrm{at}\_\ell_5[j] \vee \mathrm{at}\_\ell_6[j] \vee \mathrm{at}\_\ell_7[j]$
then goto ℓ_4
else goto ℓ_5
ℓ_4: **await** $\exists_n j : \mathrm{at}\_\ell_5[j] \vee \mathrm{at}\_\ell_6[j] \vee \mathrm{at}\_\ell_7[j]$
ℓ_5: **await** $\forall_n j : \mathrm{at}\_\ell_1[j] \vee \mathrm{at}\_\ell_2[j] \vee \mathrm{at}\_\ell_5[j] \vee \mathrm{at}\_\ell_6[j] \vee \mathrm{at}\_\ell_7[j]$
ℓ_6: **await** $\forall_n j : j < i \rightarrow (\mathrm{at}\_\ell_1[j] \vee \mathrm{at}\_\ell_2[j] \vee \mathrm{at}\_\ell_4[j])$
ℓ_7: $\langle$ *criticalsection* $\rangle$; goto ℓ_1

図 29: 公平単項的遷移系による Szymanski の排他制御アルゴリズムの記述

	Vertical View						
	P_0	$\cdots$	P_i	$\cdots$	P_{l-1}		
b_1	0	$\cdots$	0	$\cdots$	0	$\emptyset$	Horizontal View
b_2	0	$\cdots$	1	$\cdots$	1	$[i, l-1]$	

図 30: $\mathcal{P}_l$ の状態のベクトル表現

例 11.2 例 11.1 で紹介した Szymanski のアルゴリズムを WS1S 遷移系に変換するために、 2 階変数 P および $\mathrm{At}\_\ell_1, \ldots, \mathrm{At}\_\ell_7$ を用意する。初期条件 は次のようになる。

$$\exists n\colon P = \{0, \ldots, n-1\} \wedge \bigcup_{1 \le l \le k} B_l \subseteq P \wedge \forall_P i\colon (i \in \mathrm{At}\_\ell_1 \wedge \bigwedge_{2 \le l \le 7} i \notin \mathrm{At}\_\ell_l).$$

ここで、 $\forall_P j\colon f$ は $\forall j\colon j \in P \rightarrow f$ の略記である。また、 ℓ_1 に対応する遷移は次のようになる。

$$\begin{array}{l}(\forall_P j\colon j \in \mathrm{At}\_\ell_1 \cup \mathrm{At}\_\ell_2 \cup \mathrm{At}\_\ell_4) \wedge \forall_P j\colon j \neq i \rightarrow \bigwedge_{1 \le l \le 7} j \in \mathrm{At}\_\ell_l \leftrightarrow j \in \mathrm{At}\_\ell'_l \\ \quad \wedge i \in \mathrm{At}\_\ell_1 \wedge i \notin \mathrm{At}\_\ell'_1 \wedge i \in \mathrm{At}\_\ell'_2 \wedge \bigwedge_{3 \le l \le 7} i \in \mathrm{At}\_\ell_l \leftrightarrow i \in \mathrm{At}\_\ell'_l.\end{array} \tag{20}$$

■

ツール PAX の入力は WS1S 遷移系であり、 ユーザは、 検証したい MPS を、 上記の方法に従って、 WS1S 遷移系に変換しなければならない。

(2) 性質の記述

PAX における性質は、 LTL 論理式で記述される性質である。例 11.1 の Szymanski の排他制御アルゴリズムの排他性を意味する安全性は次のように記述できる。

$$\mathbf{G}\neg\exists_n i, j\colon i \neq j \wedge \mathrm{at}\_\ell_7[i] \wedge \mathrm{at}\_\ell_7[j].$$

ここで、 $\exists_n j\colon f$ は $\exists j\colon j \le n \rightarrow f$ の略記である。

11.3 抽象化

PAX における抽象化は、 前節の STeP の抽象化で説明したアサーションに基づく抽象化であり、その正当性は定理 10.2によって保証される。抽象化のもととなるアサーションの集合である基礎は、

ユーザが与えるものはなく、 MPS から変換された WS1S 遷移系の形に依存してヒューリスティックを用いて定義されている。 WS1S 論理式の充足可能性は決定可能であり、PAX は与えられた WS1S 遷移系の抽象システムを自動的に生成する。その過程で、 WS1S 論理式の充足可能性を決定するツールとして、MONA [23] が用いられる。

ヒューリステックを用いて抽象化を与えるために、次のように制限された WS1S 遷移系の遷移を考える。

$$\exists_P i\colon G \wedge L(i) \wedge C(i) \wedge V' = exp(\mathcal{V}, \mathcal{V}'). \tag{21}$$

ここで、 $G, L(i), C(i)$ は、 自由変数として $\mathcal{V}$ だけが現れる WS1S 論理式であり、 それぞれ、 G は 1 階の変数が自由に現れない論理式で、 大域条件 (global condition) を表す。 例えば、 例 11.2 の (20) 式では、 $(\forall_P j\colon j \in \mathrm{At}\_\ell_1 \cup \mathrm{At}\_\ell_2 \cup \mathrm{At}\_\ell_4)$ に対応する。 $L(i)$ は限量子をもたない論理式で、 1 階の自由変数として i だけをもつ。 直感的には、 局所条件 (local condition) を表す。 $C(i)$ は i の文脈 (context) を表す論理式で、 1 階の自由変数として i だけをもち、 1 階の変数に対する限量子を含んでもよいものである。 最後に、 $V' = exp(\mathcal{V}, \mathcal{V}')$ は、 次の状態の条件を表す。 次に、 遷移が (21) の形をした WS1S 遷移系 $\mathcal{S} = (\mathcal{V}, \Theta, \mathcal{T})$ に対する抽象化 $(\mathcal{V}_A, \Theta_A, \mathcal{T}_A)$ を与える。 まず、 述語変数の集合 $\mathcal{V}_A$ は、(21) 式の形に従って次のように決められる。

$$\begin{aligned}
\mathcal{V}_A \quad = \quad & \{b_X \mid X \in \mathcal{V}\} \cup \\
& \{b_G, b_L \mid G \text{ は大域条件}, L \text{ は局所条件 }\} \cup \\
& \{b_{L,C} \mid L \text{ は局所条件}, C \text{ は文脈 }\} \cup \\
& \{b_\xi \mid \xi \text{ は検証したい LTL 論理式に現れる論理式 }\}.
\end{aligned}$$

また、 それぞれの真偽値は次のように決められる。

$$\begin{aligned}
b_X &\equiv X \neq \emptyset \\
b_G &\equiv G \\
b_L &\equiv \exists_P i\colon L(i) \\
b_{L,C} &\equiv \exists_P i\colon L(i) \wedge C(i) \\
b_\xi &\equiv \xi
\end{aligned}$$

例 11.3 例 11.2 で得られた WS1S 遷移系に対する抽象化は、 変数として ψ_i $(1 \leq i \leq 7)$、 ϕ、 ξ をもつ。ψ_i は変数 At_ℓ_i に対応し、 その値は

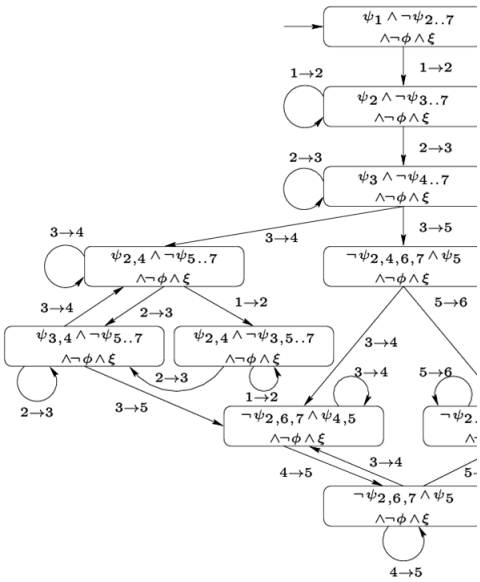
$$\psi_i \equiv \mathrm{At}\_\ell_i \neq \emptyset \quad (1 \leq i \leq 7)$$

と定義される。文脈を表す ϕ および検証したい性質に対応する ξ の値はそれぞれ

$$\begin{aligned}
\phi &\equiv \exists i\colon i \in \mathrm{At}_\ell_7 \wedge \forall j < i\colon j \in \mathrm{At}_\ell_1 \cup \mathrm{At}_\ell_2 \cup \mathrm{At}_\ell_4 \\
\xi &\equiv \neg\exists l, j\colon l \neq j \wedge l \in \mathrm{At}_\ell_7 \wedge j \in \mathrm{At}_\ell_7
\end{aligned}$$

と定義される。 ∎

以上で抽象化の定義は終わりだが、 実際には PAX は、 すべての抽象状態を求めることはせず、到達可能な状態だけを表す状態グラフ (state graph) とよばれるグラフを求めることにより、 抽象化を与える。 抽象システムに対するモデル検査は、 実際には、 状態グラフに対して行う。例 11.3で示した抽象化による状態グラフを図 31 に示す。 ここで、 各頂点は、 WS1S 論理式で表される状態の集合、 各辺は遷移関係を表す。



出典: [5]

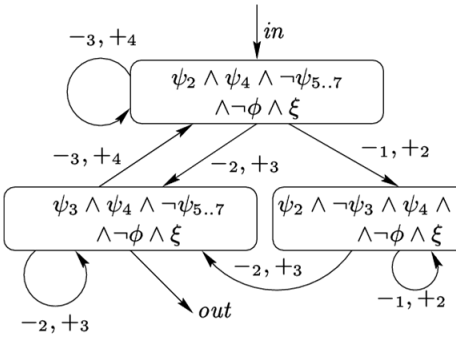
図 31: 状態グラフ

11.4 抽象化による活性の検証

PAX の特徴のひとつに、 抽象化によって活性の検証ができる場合があることがあげられる。 一般に、 ある遷移系を抽象化すると、 その具体遷移系の遷移を反映していないループができてしまい、それらのループによって活性などの性質の検証ができなくなることがある。 PAX では、 得られた抽象遷移系に対し、もとの具体遷移系から得られる情報によって公平性を仮定し、 具体遷移系でループになっていないループをなくすことができる場合がある。 例えば、例 11.1 で、 任意のプロセスが、 クリティカルセクション (ℓ_7 行目) を無限回実行できることを表す活性は次のように LTL 論理式で記述できる。

$$\phi \equiv \mathbf{GF}\psi_7 \tag{22}$$

図 31 で示した状態グラフ (G とする) を用いて (22) 式を検証する手順を以下に述べる。 まずはじめに、示したい性質に現れるすべての原始述語 ξ に対し、 G のすべての頂点 p (G の頂点は状態の集合を表す論理式である) について、 $p \to \xi$ または $p \to \neg\xi$ が成り立つか否かを調べる。 すべての頂点に対し、 どちらかが成り立つような ξ に対し、 新たな命題変数 P を導入し、 $p \to \xi$ なる頂点に対し、 新たに P でラベル付けする ($\check{G}$ とする)。 また、 検証したい活性を表す ϕ に現れるそれぞれの原始命題 ξ を、 上記の P で置き換えたものを $\check{\phi}$ と書く。(22) 式の ψ_7 に対応する新しい原始述語を P とすると、 図 31で網掛けになっている頂点が P でラベル付けされているものである。 このとき、$\check{\phi} \equiv \mathbf{GF}P$ は成り立たないが、 これは、具体システムであるもとの WS1S 遷移系の遷移を反映していない。以下のように、 $\check{G}$ をある種の詳細化を行うことによって、 活性の検証を行う。 まず、 $\check{G}$ の任意の辺 $e = (p, q)$ (p と q は頂点) と、 もとの WS1S 遷移系の任意の変数 X について、p での X の個数と q での X の個数を調べ、 減っていれば $-_X$ を、 変わらなければ $=_X$ を、 それ以外は $+_X$ を e に対しラベル付けする。さらに、 任意の辺 e と任意のラベル $-_X$ に対し、「$-_X$ でラベル付けされた e について、 e を無限回通るときは、$+_X$ でラベル付けされた辺も無限回通らければならない」という強い公平性を仮定すれば、 活性が検証できる場合がある。例えば、 図 31の状態グラフに対し上記のラベル付けをした状態グラフの一部を図 32 に示す。このように構成された



出典: [5]

図 32: ラベル付けされた状態グラフの一部

状態グラフに対して、$\check{\phi}$ が成り立つか否かをモデル検査によって調べることができ、今回は成り立つことがわかる。このことによって、(22) 式で表された活性が検証できたことになる。

11.5 適用例

文献 [5] には、本稿で紹介した Szymanski の排他制御アルゴリズムのほかに、いくつかの通信プロトコルなどの例を検証したことが報告されているが、産業界での応用例はまだないようである。文献 [5] によると、Szymanski の排他制御アルゴリズムの抽象システムを求めるために 7 分 28 秒かかるとのことである。

謝辞

産業技術総合研究所システム検証研究ラボの西澤弘毅氏は、誤りの訂正を含む重要なコメントをくださいました。また、同所属の高木理氏からは、本報告の元となったセミナーで貴重なコメントをいただきました。記して感謝します。

参考文献

[1] K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters* 22(6), pp. 307–309, 1986.

[2] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. *SPIN 2001, Workshop on Model Checking of Software*, LNCS 2057, pp. 103–122, May 2001.

[3] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. *POPL 2002*, pp. 1–3, January 2002.

[4] C. Barrett, D. Dill and J. Levitt. Validity checking for combinations of theories with equality. *Formal method in computer-aided design*, LNCS 1166, pp. 187–201, 1996.

[5] K. Baukus, S. Bensalem, Y. Lakhnech and K. Stahl. Abstracting WS1S systems to verify parameterized networks. *Proc. of TACAS'00*, pp. 188–203, 2000.

[6] N. Bjørner, A. Browne, M. Colón, B. Finkbeiner, Z. Manna, H. B. Sipma and T. E. Uribe. Verifying temporal properties of reactive systems: a STeP tutorial. *Formal Methods in System Design* 16(3), pp. 227–270, 2000.

[7] G. Brat, K. Havelund, S. Park and W. Visser. Java PathFinder – a second generation of a Java model checker. *Workshop on Advances in Verification*, 2000.

[8] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino and E. Poll. An overview of JML tools and applications. *Proc. of 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS '03)*, pp. 73–89, Electronic Notes in Theoretical Computer Science 80, Elsevier, 2003.

[9] J. R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math.* 6, pp. 66-92, 1960.

[10] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. Counterexample-Guided Abstraction Refinement. *Conference on Computer Aided Verification CAV 2000*, LNCS 1855, pp. 154–169, 2000.

[11] E. M. Clarke, O. Grumberg and D. E. Long. Model Checking and Abstraction. *ACM-TOPLAS* 16(5), pp. 1512–1542, 1994.

[12] E. M. Clarke, O. Grumberg and D. O. Peled. *Model Checking*, MIT Press, 1999.

[13] M. A. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. *Proc. of CAV'98*, LNCS 1427, pp. 293–304, 1998.

[14] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. *Proceedings of the 22nd International Conference on Software Engineering*, pp. 439–448, 2000.

[15] D. L. Detlefs, G. Nelson and J. B. Saxe. A theorem prover for program checking. *Research Report* 178, Compaq SRC, 2002.

[16] E. W. Dijkstra. *A discipline of Programmig*, Prentice Hall, Englewood, Cliffs, NJ, 1976.

[17] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe and R. Stata. Extended static checking for Java. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* volume 37(5) of SIGPLAN Notices, pp. 234–245, 2002.

[18] M. M. Gallardo, J. Martinez, P. Merino and E. Pimentel. αSPIN: Extending SPIN with Abstraction. *Proc. of the 9th SPIN Workshop*, pp. 254–258, 2002.

[19] M.M. Gallardo and P. Merino. A Framework for Automatic Construction of Abstract Promela Models. *Theoretical and Practical Aspects of Spin Model Checking*, LNCS 1680, pp. 184–199, Springer, 1999.

[20] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. *Conference on Computer Aided Verification CAV'97*, LNCS 1254, pp. 72–83, 1997.

[21] J. Hatcliff. Presentation slides for a talk in ICSE'2000.

[22] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *International Journal on Software Tools for Technology Transfer* Vol. 2, No. 4, pp. 366–381, 2000.

[23] J. G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe and A. Sandholm. MONA: Monadic second-order logic in practice. *TACAS'95*, LNCS 1019, pp. 89–110, 1996.

[24] T. A. Henzinger, R. Jhala, R. Majumdar and G. Sutre. Lazy Abstraction. *ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pp. 58–70, 2002.

[25] T. A. Henzinger, R. Jhala, R. Majumdar and G. Sutre. Software Verification with Blast.

Proceedings of the 10th SPIN Workshop on Model Checking Software, LNCS 2648, pp. 235–239, 2003.

[26] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre and W. Weimer. Temporal-Safety Proofs for Systems Code. *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV)*, LNCS 2404, pp. 526–538, 2002.

[27] G. J. Holzmann. *The SPIN Model Checker*, Addison-Wesley, 2003.

[28] G. J. Holzmann and M. H. Smith. Automating Software Feature Verification. *Bell Labs Technical Journal* 5(2), pp. 72–87, 2000.

[29] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety*, Springer-Verlag, New York, 1995.

[30] K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. ESC/Java User's Manual. *Technical Note* 2000-002, Compaq Systems Research Center, 2000.

[31] K. Rustan, M. Leino, J. B. Saxe and R. Stata. Checking Java programs via guarded commands. *SRS Technical Note* 1999-002, Compaq Systems Research Center, 1999.

[32] B. K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. *Proc. of International Conference on Supercomputing Systems 1988*, pp. 621–626, 1988.

[33] W. Thomas. Automata on infinite objects. *Handbook of theoretical computer science, volume B: formal method and semantics*, pp. 134–191, Elsevier Science Publishers, 1990.

[34] R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam and V. Sundaresan. Soot – a Java bytecode optimization framework. *Proceedings of CASCON '99*, pp. 125–135, 1999.

[35] W. Visser, K. Havelund, G. Brat and S. Park. Model Checking Programs. *Proc. of International Conference on Automated Software Engineering*, pp. 3–11, 2000.

[36] PVS – `http://pvs.csl.sri.com/`

[37] Simplify – `http://research.compaq.com/SRC/esc/Simplify.html`

[38] Spin – `http://spinroot.com/spin/`

[39] Vampyre – `http://www-cad.eecs.berkeley.edu/~rupak/Vampyre/`