統一的プログラム表現にもとづく統合的ソフトウェア開発支援環境の構築に関する研究

笠原, 義晃
九州大学工学研究科情報工学専攻

https://doi.org/10.11501/3110903

出版情報：九州大学, 1995, 博士（工学）, 課程博士
バージョン：
権利関係：

統一的プログラム表現にもとづく
統合的ソフトウェア開発支援環境の
構築に関する研究

笠原義晃

1996 年 1 月

目次

図一覧

表一覧

第 1 章

はじめに

1.1 背景

近年、 マルチプロセッサシステムや、 LAN によるワークステーションクラスタの普及が進んでいる。単一の CPU の処理能力には自ずと限界があり、 この限界を越えた処理を行うためには、必然的に複数の CPU による並列処理、複数の計算機による分散協調処理が必要となるためである。今後、計算機システムの処理能力に対する要求が高まるにつれ、システムはより大規模になっていくと思われる。当然、このようなシステムで動作する大規模並行プログラムに対する需要も今後ますます増加していくはずである。信頼性の高い並行処理システムを構築するためには、並行処理システムにおいて中心的な役割を果たす並行プログラムの信頼性を高めることが重要な課題である。しかし、現在のところ、並行プログラムの信頼性を高める手法はさかんに研究されているものの実用的に見て十分ではなく、現実にプログラムのバグのためにさまざまな事故が起こっている。例を挙げると、湾岸戦争で活躍した米軍のパトリオットミサイルに搭載された追尾システムにはバグがあったため、イラク軍のスカッドミサイルを (最初予期されたほど) 撃墜できなかったし、スペースシャトル等もソフトウェアの不具合のために打ち上げ延期や作戦の失敗を何度も起こしている [30]。また身近な所では銀行のオンラインシステムの不具合のせいでキャッシュディスペンサーが使えなかった経験のある人もあろう。

並行プログラムでは一般に複数のプロセスがそれぞれまちまちの速度で動作している。このため、並行プログラムの内部には多重の制御の流れとデータの流れが同時に存在し、それらが非決定的に相互作用している。逐次プログラムが、同じ入力を与えれば同じ動作をするのに対

し、並行プログラムはこの非決定性により、同じ入力を与えたとしても実行のたびに異なる動作をする可能性がある。このように並行プログラムには逐次プログラムにない性質があるため、逐次プログラム開発の方法論をそのまま流用するのでは、並行プログラムに特有の問題を解決することはできない。しかし、現在の所効果的な並行プログラム開発手法がはっきりわかっていないこともあって、並行プログラムの開発時にも逐次プログラムの開発に用いられているツールや方法論の流用が行われている。このため、並行プログラムの信頼性は十分に確保できていない。形式的な手法を用いて、プログラムの正しさを数学的に証明するという方法も研究されているが、現在までのところまだ実用化するには至っていない。われわれは、信頼性の高い並行プログラムを効率よく開発するためには、プログラム開発のさまざま場面において人間の能力を補助しソフトウェアの信頼性を高める手助けをする開発支援環境の存在が重要であると考える。もちろんこの開発支援環境は並行プログラムの特徴を考慮しているものでなければならない。

さまざまな開発活動において、それぞれ異なる支援環境を用いる必要があるのでは効率が悪いし、それぞれの開発活動の間の連携にも問題が出る。したがって、開発支援環境は複数の開発活動を統合的に支援できることが望ましい。また、大規模なソフトウェアでは、それぞれの構成部分においてその分野を処理するのに有利な、異なる言語で記述されることも多い。この時、言語ごとに異なる開発環境を用いるのではやはり効率が悪い。したがって、開発支援環境はさまざまなプログラミング言語で書かれたプログラムの開発作業を統一的に支援できるようなものであることが望ましい。

さまざまな開発活動を統合的に支援する開発環境を構築するために、われわれは、それぞれの活動を支援する複数の支援ツール群と、これを統合し単一のツールとして提供するユーザインターフェイスを用いることを考えた。それぞれのツールは単体または複数の組み合わせで利用でき、ユーザが新しい開発支援環境を作るのに利用できる部品として使える。また、これらの支援ツールを複数の開発言語に対応させるためには、言語に依存せず、ソフトウェアの開発活動において多くの応用が可能な、統一的なプログラムの抽象表現にもとづいて実装されていることが望ましいと考える。ソフトウェア開発・保守活動の際には、対象プログラムから必要な情報を取り出し、また不必要な情報を隠蔽するために、なんらかの抽象表現が必要になることが多い。それによって、余分な情報による不必要な混乱を防ぎ、また対象プログラムのさまざまな特徴を明確にすることができる。また、開発支援環境側から見た場合、対象プログラム

一旦解析して抽象表現として内部に保持することは、環境内のさまざまなツールにおける対象プログラム解析の負担の軽減と、ツール間の情報の共有に有用である。

対象言語に依存せず、ソフトウェアの開発活動において多くの応用が可能なプログラムの抽象表現として、プログラム従属性にもとづくモデルがある。プログラム従属性とは、プログラム中の制御の流れとデータの流れによって決定される、プログラムの各文間に存在する従属関係のことである。例えば、ある変数の値を定義している文は、そこでその変数の値を決定するために参照している他の変数の値を定義している文から影響を受ける。また、ある条件分岐文の選択枝となっている文はその条件分岐文の評価に影響を受ける。このように、プログラム内のある文の実行によってその後に実行される別の文の実行がなんらかの影響を受けるとき、後者は前者に従属していると言う。この関係はある意味で命令型プログラムの振舞いの本質を示していると考えることができる。

命令型逐次プログラムに関しては、制御フローグラフ (CFG)、定義使用グラフ (DUG)[3][34]、プログラム従属グラフ [19][23][29] といったいくつかの抽象表現が提案されている。これらの表現モデルには、コンパイラ構築、最適化、プログラムの並列化、理解、テスト、デバッグ、複雑さ計測等のソフトウェア開発・保守活動においてさまざまな応用が可能であることが知られており、また実際に利用されている [3][23][32][33][34]。

これに対し、われわれはすでに命令型逐次・並行プログラムの表現モデルとして、非決定的並列制御フローネット (CFN)、非決定的並列定義使用ネット (DUN)[18] およびプロセス従属ネット (PDN)[12][13][14] を提案している。これらは枝が分類された有向グラフである。 CFN は並行プログラム中の複数の制御の流れを表現する。 DUN は並行プログラム中の複数の制御の流れと、変数の定義と使用、そしてプロセス間の同期・通信を表現する。 PDN は並行プログラム中のプログラム従属性を明示的に表現する。これらの表現モデルは逐次プログラムにおける CFG、 DUG、 PDG を拡張した形であるため、逐次・並行プログラムの開発・保守に関するさまざまな応用分野があると考えられる [14][15][17]。

統合的開発支援環境に関する現在の状況を見てみると、逐次プログラムの開発支援を行う環境は、統合化 CASE といった形ですでに多数実用化され、実際に販売されている。この中には、プログラム従属性を利用した支援を行えるものも存在する [37]。しかし、これらの環境は基本的に逐次プログラムのためのものであり、並行プログラムの開発時には必ずしも有効ではない。並行プログラムの信頼性をより高めるため、上流 CASE にはじまって、エディタ、コン

パイラ、テスト検証ツール、デバッガ等並行処理ソフトウェアの開発支援に関する研究が数多くなされてきているが、これらのほとんどはある特定のプログラミング言語による開発活動のうちのある特定の部分のみに注目している。並行プログラムの開発を統合的に支援する環境の構築を目指した研究としては、主に Ada を対象言語として研究が進められた Arcadia プロジェクト [24] や SLCSE(Software Life Cycle Support Environment)[36] がある。しかし、並行プログラムのプログラム従属性を利用した開発支援環境に関する研究は今までなかった。

1.2 目的

本研究では以上の背景にもとづいて、主にプログラム従属性にもとづいた統一的プログラム表現を用いて、さまざまな命令型プログラミング言語で記述された逐次・並行プログラムの開発を統一的に支援する、統合的な開発支援環境の構築を目的とする。

これを達成するためには、以下のような問題を解決する必要がある。

- 対象プログラムのソースリストから統一的なプログラム表現への変換手法と、これを実装した変換ツールの開発
- 統一的プログラム表現を用いた各種ソフトウェア開発支援手法の考案
- 各種ソフトウェア開発支援を行うツールの実装とその統合

1.3 成果

本研究の主な成果は以下の通りである。

- CFN/DUN および PDN 生成ツールの開発

  C、Pascal、および Ada で記述されたプログラムを入力とし、対象プログラムの CFN/DUN を自動生成するツールを開発した。また、CFN/DUN から PDN を生成するアルゴリズムを考案し、これを実装して CFN/DUN から PDN を自動生成するツールを開発した。これについては第 3章で述べる。

- 統一的プログラム表現にもとづくソフトウェア開発支援ツールの試作とその統合

本論文で提案した統一的プログラム表現にもとづいて、スライス生成ツール、グラフ / ネット可視化ツール、およびソースリスト編集 / 表示ツールの試作を行った。これについては第 4章で述べる。

- Ada83 並行プログラムのデッドロック検出ツールの開発

  統合的開発支援ツール開発の一環として、 Ada83 並行プログラムのデッドロック検出ツールを開発した。デッドロックは並行プログラムにおける重要な問題であるため、特に重点的な研究を行った。このツールは Ada83 並行プログラム中に発生する可能性のある 18 種類のデッドロック全てを、対象プログラムの実行時に動的に検出することができる。これについては第 5章で述べる。

本論文は 6章から構成される。これ以降の構成は次の通りである。

第 1 章は序論であり、研究の背景、動機およぼ目的について述べた。

第 2 章では、本研究で構築する環境で逐次・並行プログラムの抽象表現モデルとして用いる、 CFN、 DUN、および PDN を定義する。まずこれらを定義するために必要なグラフ理論上の概念について定義する。次にこれらを用いて CFN、 DUN、および PDN を定義する。これらは枝が分類された有向グラフである。 CFN は並行プログラム中の複数の制御の流れを表現する。 DUN は並行プログラム中の複数の制御の流れと、変数の定義と使用、そしてプロセス間の同期・通信を表現する。 PDN は並行プログラム中のプログラム従属性を明示的に表現する。これら 3 つのモデルは、逐次プログラムの抽象表現モデルである CFG、 DUG、および PDG を拡張したものである。従って、逐次プログラムの開発支援においてこれらのモデルがさまざまな応用を持つのと同様、 CFN、 DUN、および PDN は並行プログラムの開発支援においてやはりさまざまな応用が考えられるはずである。このような応用分野についてもこの章で述べる。また、これらのモデルを実際に開発支援に用いる際に解決しなければならない問題点について述べる。

第 3 章では、第 2 章で述べた CFN、 DUN、および PDN を対象プログラムから自動生成する手法と、これを実装したツールについて述べる。 CFN、 DUN、および PDN を実際に並行プログラムの開発支援に利用するためには、自動生成の手法およびそれを実装したツールが不可欠である。本研究では、プログラミング言語 Ada、 C、 Pascal で書かれたプログラムのソースリストから、これらのモデルを生成する手法の考案とツールの開発を行ったが、この論

文では、特にプログラミング言語 Ada で書かれた並行プログラムを対象として、対象プログラムからの CFN、 DUN、および PDN の自動生成アルゴリズムと、これを実装したツールの説明を行う。

第 4 章では、第 2 章で述べた CFN、 DUN、および PDN にもとづく統合的なソフトウェア開発支援環境の構築について述べる。まずこれらのモデルが開発支援環境のための抽象表現として適していることを述べる。ソフトウェアの開発や保守の際には、プログラマは対象プログラムを読んで動作を理解する必要がある。この時に重要なプログラムの実行文間のさまざまな関係が CFN、 DUN、および PDN には明示的に示されている。また、これらのモデルは対象プログラミング言語に依存しない形で定義されており、複数の言語を用いて開発されているシステムの開発支援にも利用することができる。第 3 章で述べたように CFN、 DUN、および PDN はソフトウェア開発支援におけるさまざまな応用分野がある。以上のことから、これらのモデルをもとに統合的なソフトウェア開発支援環境を構築することができると考える。次に、このモデルにもとづいて現在開発している統合的開発支援環境の概要について述べる。現在一部のツールが実装され、統合されつつある。現在統合されている部分について、画面の例とともに紹介し、これらのツールをソフトウェアの開発活動においてどのように利用できるかを述べる。

第 5 章では、本環境に統合を予定している Ada 並行プログラムのための動的デッドロック検出ツールについて述べる。まず、 Ada83 のデッドロック検出について述べる。 Ada83 のタスク間には 5 種類の待機状態によって 18 種類のデッドロックが形成される可能性がある。本研究で開発したタスキングデッドロック検出ツールは、 Ada83 並行プログラムの実行を監視し、タスク待ちグラフを生成し、これに発生したループを検出することにより、これら 18 種類のデッドロックを全て検出することができる。次に Ada83 の新版である Ada95 に対応するために、 Ada95 で追加されたタスキング機構について考察する。

第 6 章は、本論文のまとめである。

第 2 章

並行プログラムの表現モデル

この章では、本研究で並行プログラムの抽象表現として用いる、 CFN、 DUN、 PDN を定義する。またそれらのモデルの応用の可能性について述べる。

2.1　用語の定義

この章では、本論文で CFN、 DUN および PDN を定義するために用いる用語の定義を行う。

定義 2.1.1 有向グラフとは、**節点**の有限集合 $\mathbf{V}$、 **枝**の有限集合 $\mathbf{A}(\mathbf{A} \subseteq \mathbf{V} \times \mathbf{V}$ は $\mathbf{V}$ 上の 2 項関係である) からなる順序対 $(\mathbf{V}, \mathbf{A})$ である。任意の枝 $(v_1, v_2) \in \mathbf{A}$ について、 v_1 を**始点**と呼び、 v_1 は v_2 **に隣接している**という。また v_2 を**終点**と呼び、 v_2 は v_1 **から隣接されている**という。 v に隣接している節点を v の **先行節点**、 v から隣接されている節点を v の**後続節点**という。 v の先行節点の数を**入次数**と呼び、 $in\text{-}degree(v)$ で表す。 v の後続節点の数を**出次数**と呼び、 $out\text{-}degree(v)$ で表す。有向グラフ $(\mathbf{V}, \mathbf{A})$ において、任意の $v \in \mathbf{V}$ について $(v, v) \notin \mathbf{A}$ のときこれを**単純有向グラフ**と呼ぶ。 □

定義 2.1.2 枝が分類された有向グラフ とは、各 $(\mathbf{V}, \mathbf{A}_i)$ $(i = 1, \ldots, n-1)$ が有向グラフでかつ $\mathbf{A}_i \cap \mathbf{A}_j = \emptyset$ $(i = 1, \ldots, n-1$、 $j = 1, \ldots, n-1$、 $i \neq j)$ であるような n 項組 $(\mathbf{V}, \mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_{n-1})$ である。任意の $v \in \mathbf{V}$ について、 $(v, v) \notin \mathbf{A}_i (i = 1, \ldots, n-1)$ であるような $(\mathbf{V}, \mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_{n-1})$ を**枝が分類された単純有向グラフ**と呼ぶ。 □

定義 2.1.3 有向グラフ $(\mathbf{V}, \mathbf{A})$ または枝が分類された有向グラフ $(\mathbf{V}, \mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_{n-1})$ において、$1 \leq i \leq l-1$ で a_i の終点が a_{i+1} の始点であるような枝の並び $(a_1, a_2, \ldots, a_l)$ を**経路**と呼ぶ。ここで $a_i \in \mathbf{A}(1 \leq i \leq l)$ または $a_i \in \mathbf{A}_1 \cup \mathbf{A}_2 \cup \ldots \mathbf{A}_{n-1}(1 \leq i \leq l)$ であり、$l(l \geq 1)$ をその経路の**長さ**と呼ぶ。a_1 の始点を v_I、a_l の終点を v_T とするとき、この経路を v_I から v_T への経路、または単に経路 v_I-v_T と呼ぶ。 □

2.2　制御フローネット (CFN) と定義使用ネット (DUN)

この章では、CFN と DUN について述べる。

定義 2.2.1 非決定的並行制御フローネット (以下 CFN) は 10 項組 $(\mathbf{V}, \mathbf{N}, \mathbf{P}_F, \mathbf{P}_J, \mathbf{A}_C, \mathbf{A}_N, \mathbf{A}_{\mathbf{P}_F}, \mathbf{A}_{\mathbf{P}_J}, \mathbf{s}, \mathbf{t})$ で表される。ここで、$(\mathbf{V}, \mathbf{A}_C, \mathbf{A}_N, \mathbf{A}_{\mathbf{P}_F}, \mathbf{A}_{\mathbf{P}_J})$ は枝が分類された単純有向グラフで $\mathbf{A}_C \subseteq \mathbf{V} \times \mathbf{V}, \mathbf{A}_N \subseteq \mathbf{N} \times \mathbf{V}, \mathbf{A}_{\mathbf{P}_F} \subseteq \mathbf{P}_F \times \mathbf{V}, \mathbf{A}_{\mathbf{P}_J} \subseteq \mathbf{V} \times \mathbf{P}_J$ であり、$\mathbf{N} \subset \mathbf{V}$ は**非決定的選択点**と呼ばれる節点の集合、$\mathbf{P}_F \subset \mathbf{V}(\mathbf{N} \cap \mathbf{P}_F = \emptyset)$ は**並行実行分岐点**と呼ばれる節点の集合、$\mathbf{P}_J \subset \mathbf{V}(\mathbf{N} \cap \mathbf{P}_J = \emptyset)$ は**並行実行合流点**と呼ばれる節点の集合、$\mathbf{s} \in \mathbf{V}$ は**開始点** と呼ばれる $in\text{-}degree(s) = 0$ であるような唯一の節点、$\mathbf{t} \in \mathbf{V}$ は**終了点**と呼ばれる $out\text{-}degree(t) = 0$ かつ $\mathbf{t} \neq \mathbf{s}$ であるような唯一の節点で、かつ任意の $v \in \mathbf{V}(v \neq \mathbf{s}, v \neq \mathbf{t})$ について $\mathbf{s}$ から v へと v から $\mathbf{t}$ へそれぞれ少なくとも 1 つの経路がある。任意の枝 $(v_1, v_2) \in \mathbf{A}_C$ を**逐次制御枝**、任意の枝 $(v_1, v_2) \in \mathbf{A}_N$ を**非決定的選択枝**、任意の枝 $(v_1, v_2) \in \mathbf{A}_{\mathbf{P}_F} \cup \mathbf{A}_{\mathbf{P}_J}$ を **並行実行枝**と呼ぶ。 □

定義 2.2.2 CFN に含まれる任意の 2 つの節点を u, v とする。v から 終了点 $\mathbf{t}$ への全ての経路が u を含むときかつそのときに限り、u は v を**前方支配する (forward dominate)** という。u が v を前方支配し、かつ $u \neq v$ のときかつそのときに限り、u は v を**真に前方支配する (properly forward dominate)** という。u が v を前方支配し、かつ v からの長さ k 以上の全ての経路が u を含むような整数 $k(k \geq 1)$ が存在するときかつそのときに限り、u は v を**強く前方支配する (strongly forward dominate)** という。u が v を真に前方支配し、かつ v から u への全ての経路が、v を真に前方支配する他の節点を含まないときかつそのときに限り、u を v の**直接前方支配点 (immediate forward dominator)** と呼ぶ。v から u への経路に含まれる全ての節点が v を前方支配し、かつ u の後続節点が v を前方支配しないときか

つそのときに限り、 u を v の**最終連続前方支配点 (last continuous forward dominator)** と呼ぶ。 □

定義 2.2.3 非決定的並行定義使用ネット (以下 DUN) は 7 項組 $(\mathbf{N}_C, \Sigma_V, \mathbf{D}, \mathbf{U}, \Sigma_C, \mathbf{S}, \mathbf{R})$ で表される。ここで $\mathbf{N}_C = (\mathbf{V}, \mathbf{N}, \mathbf{P}_F, \mathbf{P}_J, \mathbf{A}_C, \mathbf{A}_N, \mathbf{A}_{\mathbf{P}_F}, \mathbf{A}_{\mathbf{P}_J}, \mathbf{s}, \mathbf{t})$ は CFN、 Σ_V は **変数**と呼ばれるシンボルの有限集合、 $\mathbf{D} : \mathbf{V} \to P(\Sigma_V)$ と $\mathbf{U} : \mathbf{V} \to P(\Sigma_V)$ はそれぞれ $\mathbf{V}$ から Σ_V の冪集合への部分関数、 Σ_C は **チャネル**と呼ばれるシンボルの有限集合、 $\mathbf{S} : \mathbf{V} \to P(\Sigma_C)$ と $\mathbf{R} : \mathbf{V} \to P(\Sigma_C)$ はそれぞれ $\mathbf{V}$ から Σ_C の冪集合への部分関数である。 □

従来の (決定的で逐次的な)CFG は、定義 2.2.1で述べた CFN から並行的な要素を除いた特殊な場合と考えられる。このとき、 $\mathbf{N}, \mathbf{P}_F, \mathbf{P}_J, \mathbf{A}_N, \mathbf{A}_{\mathbf{P}_F}, \mathbf{A}_{\mathbf{P}_J}$ はそれぞれ空集合である。従って、この CFN は、 C、 Pascal、 Ada、 Occam 2 といった手続き型プログラミング言語で書かれた逐次・並行プログラムの、単一・複数の制御フローを統一的に表現するために利用できる。

定義 2.2.2で述べた、 CFN の節点間の関係に関する用語の定義は従来 CFG について研究されてきたものと基本的に同じである [34]。

通常の (決定的で逐次的な)DUG は、定義 2.2.3で述べた DUN から並行的な要素を除いた特殊な場合と考えられる。このとき、 $\mathbf{N}, \mathbf{P}_F, \mathbf{P}_J, \mathbf{A}_N, \mathbf{A}_{\mathbf{P}_F}, \mathbf{A}_{\mathbf{P}_J}, \Sigma_C, \mathbf{S}, \mathbf{R}$ はそれぞれ空集合である。 DUN は CFN に変数の定義と使用、および同期 / 通信のためのチャネルに関する情報を付加したものと考えられる。

上記の CFN と DUN の定義はグラフ理論的なものであり、どんなプログラミング言語とも独立である。

DUN を手続き型並行プログラムに適用すると、プログラムの制御フローを有向枝で表し、実行文、制御論理式、およびプロセス間の同期点を節点で表すような、枝が分類された有向グラフに、さらに各節点での変数の定義、使用の状況、およびプロセス間の同期や通信の状況をラベルとして節点に付加したラベル付き有向グラフとなる。

本論文では、図 2.1に載せた Ada 並行プログラム [40] を例題として用いる。このプログラムには、手続き `Sample`(仕様が 3 行～ 52 行、本体が 55 行～ 62 行) を実行する環境タスク (*environment task*) と、その子タスクであるタスク `PV`(仕様が 9 行～ 12 行、本体が 18 行～ 34 行) とタスク `Monitor`(仕様が 14 行～ 16 行、本体が 36 行～ 52 行) の 3 つのタスクが含まれて

```
 1  with Text_IO;
 2
 3  procedure Sample is
 4
 5  package Int_IO is new Text_IO.Integer_IO (Integer);
 6
 7     Value : Integer;
 8
 9  task PV is
10     entry Read (X : out Integer);
11     entry Add (X : in Integer);
12  end PV;
13
14  task Monitor is
15     entry Quit;
16  end Monitor;
17
18  task body PV is
19     V : Integer := 0;
20  begin
21     loop
22        select
23           accept Read (X : out Integer) do
24              X := V;
25           end Read;
26        or
27           accept Add (X : in Integer) do
28              V := V + X;
29           end Add;
30        or
31           terminate;
32        end select;
33     end loop;
34  end PV;
35
36  task body Monitor is
37     V : Integer;
38     Finish : Boolean := False;
39  begin
40     while (not Finish) loop
41        select
42           accept Quit;
43           Finish := True;
44        or
45           delay 5.0;
46           PV.Read (V);
47           if V = 0 then
48              Put_Line ("Buffer empty");
49           end if;
50        end select;
51     end loop;
52  end Monitor;
53
54
55  begin -- sample
56     loop
57        Int_IO.Get (Value);
58        exit when Value = 0;
59        PV.Add (Value);
60     end loop;
61     Monitor.Quit;
62  end Sample;
```

図 2.1: Ada で書かれたサンプルプログラム。

いる。

図 2.2に図 2.1の Ada プログラムに関する DUN を示す。環境タスクは図 2.2では手続き名 `Sample` で示している。図 2.1 の行番号が図 2.2の各節点に対応している。 D(節点) はその節点で定義されている変数、 U(節点) はその節点で使用されている変数、 S(節点) はその節点で呼び出しをかけているチャネル名、 R(節点) はその節点で呼び出しを受け付けているチャネル名

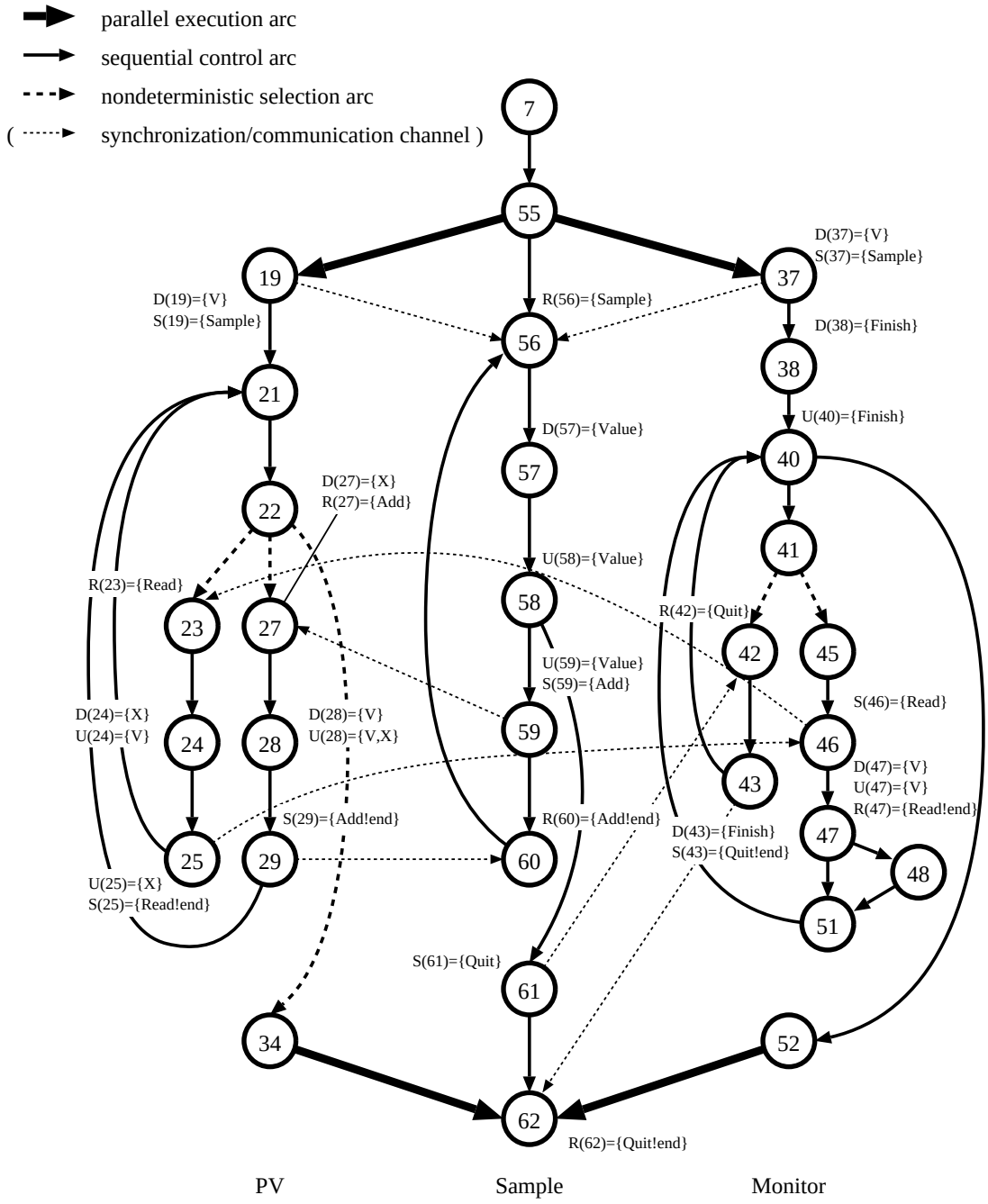

図 2.2: 図 2.1 のサンプルプログラムに関する DUN

を返す関数である。チャネルは本来枝として定義されているものではないが、対応関係が分かりやすくなるように便宜的に枝として図に描き入れた。

2.3　プロセス従属ネット (PDN)

CFN および DUN を用いることにより、並行プログラム中のさまざまな基本的従属関係を定義することができる [14]。

この章では、まず並行プログラムにおける基本的な 5 つの従属性である、制御従属性、選択従属性、データ従属性、同期従属性、通信従属性をそれぞれ定義し、プログラム上でのそれらの意味について説明する。その後、プロセス従属ネットを定義する。

定義 2.3.1 ある並行プログラムの CFN を $(\mathbf{V}, \mathbf{N}, \mathbf{P}_F, \mathbf{P}_J, \mathbf{A}_C, \mathbf{A}_N, \mathbf{A}_{\mathbf{P}_F}, \mathbf{A}_{\mathbf{P}_J}, \mathbf{s}, \mathbf{t})$ とし、このネット中に $u \in \mathbf{V}, v \in (\mathbf{V} - (\mathbf{N} \cup \mathbf{P}_F \cup \mathbf{P}_J))$ であるような任意の 2 節点 u, v をとる。 v の直接前方支配点を含まないような v から u への経路 $P = (v_1 = v, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n = u)$ が存在し、かつ v' から u への経路が v' の直接前方支配点を含まないような節点 v' が P の中にないときかつそのときに限り、 u は v に**直接強く制御従属** しているという。 v に 2 つの後続節点 v' と v'' があって、 v から u への経路 $P = (v_1 = v, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n = u)$ が存在し、 P に含まれる任意の節点 $v_i (1 \leq i \leq n)$ が v' を強く前方支配していてかつ v'' を強く前方支配していないときかつそのときに限り、 u は v に **直接弱く制御従属**しているという。 □

この制御従属の定義は [34] がもとになっているが、強い制御従属の定義が若干異なる。[34] の定義では、入れ子になった `if` 文に含まれる実行文はその文を含む全ての `if` 文の条件分岐文に制御従属しているが、定義 2.3.1ではその文を含む一番内側の `if` 文の条件分岐文にのみ制御従属していることになる。これにより、入れ子になった `if` などの制御従属性の表現が簡略化され、階層関係がより明確になる。

v には少なくとも 2 つの後続節点 v' と v'' があって、 v から v' への分岐が実行されれば u は必ず実行されるが、 v から v'' への分岐が実行されれば u は実行されてはならないとき、 u は v に直接強く制御従属している。 v には少なくとも 2 つの後続節点 v' と v'' があって、 v から v' への分岐が実行されれば u は決まったステップ数を経て実行されるが、 v から v'' への分岐が実行されると u は実行されてはならないか、もしくは実行が不定期に遅らされるとき、 u は v に直接弱く制御従属している。弱い制御従属は、ループの脱出条件文とループの外にある文の関係を示している。

定義 2.3.1にもとづくと、もし u が v に直接強く制御従属していれば、同時に u は v に直接弱く制御従属している。しかし、逆は必ずしも真ではない。強い制御従属は弱い制御従属を含

んでいるため、以後の議論では特に指定しない限り「制御従属」は弱い制御従属を表すこととする。

例えば図 2.2で、節点 59, 60 は 58 の `exit when` 文の制御述語の評価によって実行されるかどうかが決まる。従って、これらの節点は 55 に強く制御従属している。また 61 は 55 に直接弱く制御従属しているが、強くは制御従属していない。

定義 2.3.2 ある並行プログラムの CFN を $(\mathbf{V}, \mathbf{N}, \mathbf{P}_F, \mathbf{P}_J, \mathbf{A}_C, \mathbf{A}_N, \mathbf{A}_{\mathbf{P}_F}, \mathbf{A}_{\mathbf{P}_J}, \mathbf{s}, \mathbf{t})$ とし、このネット中に $u \in \mathbf{V}, v \in \mathbf{N}$ であるような任意の 2 節点 u, v をとる。ここで、 v の直接前方支配点を含まないような v から u への経路 $P = (v_1 = v, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n = u)$ が存在し、かつ v_i から u への経路が v_i の直接前方支配点を含まないような節点 $v_i (1 \leq i \leq n)$ が P の中に存在しないときかつそのときに限り、 u は v に**直接選択従属** しているという。 □

v に複数の後続節点があって、 v からある後続節点への分岐が実行されれば u が必ず実行されるが、それ以外の分岐が実行された場合は u が実行されてはならないとき、 u は v に直接選択従属している。また、ここでいう分岐は制御従属とは異なり、 $\mathbf{N}$ の要素、すなわち非決定的選択枝である。例えば図 2.2で、 22 の `select` 文では、 23 にあるエントリ `Read` と 27 にあるエントリ `Add` のどちらか先に呼び出された方に実行が移る。どちらに実行が移るかはどちらのエントリが先に呼び出されるかによって決まり、これは一般に予測不能である。従って、 23 から 29 までの各節点は 22 に直接選択従属している。

選択従属と制御従属は同一プロセス内での制御の流れの変化という点で似ている。しかし、制御従属は制御従属している文の実行が条件分岐文の評価、ひいてはプログラムへの入力によって一意に決定されるのに対し、選択従属はプログラムへの入力が同じでも複数のプロセスの実行タイミングの変化により選択肢が異なってくる可能性がある。このため、制御従属と区別している。

定義 2.3.3 ある並行プログラムの DUN を $(\mathbf{N}_C, \Sigma_V, \mathbf{D}, \mathbf{U}, \Sigma_C, \mathbf{S}, \mathbf{R})$、このネット中の任意の 2 節点を u, v とする。$(\mathbf{D}(v) \cap \mathbf{U}(u)) - \mathbf{D}(P') \neq \emptyset$(ここで $\mathbf{D}(P') = \mathbf{D}(v_2) \cup \ldots \cup \mathbf{D}(v_n - 1)$) であるような v から u への経路 $P = (v_1 = v, v_2), (v_2, v_3), \cdots, (v_{n-1}, v_n = u)$ が存在するときかつそのときに限り、 u は v に**直接データ従属**しているという。 □

この定義は [34] におけるデータフロー従属と同じものである。

v で計算されている変数の値が u で計算される変数の値に直接影響を与えるとき、u は v に直接データ従属している。例えば図 2.2で、28 で値を定義されている変数 V を例にとると、'V :=V+X;' という式から、V の値は変数 V と変数 X の値によって決定されることがわかる。V は 19 と 28 で、X は 27 で値を定義されている可能性がある。従って、28 は 19, 27, 28 に直接データ従属している。

定義 2.3.4 ある並行プログラムの DUN を $(\mathbf{N}_C, \Sigma_V, \mathbf{D}, \mathbf{U}, \Sigma_C, \mathbf{S}, \mathbf{R})$、ネット中の任意の 2 節点を u, v とする。ここで $\mathbf{N}_C$ は CFN$(\mathbf{V}, \mathbf{N}, \mathbf{P}_F, \mathbf{P}_J, \mathbf{A}_C, \mathbf{A}_N, \mathbf{A}_{\mathbf{P}_F}, \mathbf{A}_{\mathbf{P}_J}, \mathbf{s}, \mathbf{t})$ である。以下に述べる条件のうちどれかが満たされるときかつそのときに限り、u は v に**直接同期従属**しているという。

1. $(v, u) \in \mathbf{A}_{\mathbf{P}_F} \cup \mathbf{A}_{\mathbf{P}_J}$、すなわち (v, u) は並行実行枝。
2. $\mathbf{S}(v) = \mathbf{R}(u)$。
3. v' が v に直接同期従属していて、u が v' の最終連続前方支配点で、かつ v' から u への経路内の任意の節点 $v''(v'$ を除く$)$ において $\mathbf{S}(v'') = \emptyset$ かつ $\mathbf{R}(v'') = \emptyset$ であるような v' が存在する。 □

同期従属は 2 つのプロセス間での同期によって発生する従属関係である。あるプロセスにある文 v の処理の開始や終了によって別のプロセスにある文 u の処理の開始や終了が影響を受ける可能性がある場合に u は v に直接同期従属している。

例えば図 2.2で、環境タスクは本体の最初の文 56 の実行前に子タスクであるタスク PV と Monitor が起動されるのを待たなければならない。すなわち、19 および 37 の処理の開始を待って 56 の処理が開始される。従って、56 は 19 と 37 に同期従属している。

定義 2.3.5 ある並行プログラムの DUN を $(\mathbf{N}_C, \Sigma_V, \mathbf{D}, \mathbf{U}, \Sigma_C, \mathbf{S}, \mathbf{R})$、ネット中の任意の 2 節点を u, v とする。u が v' に直接データ従属していて、$\mathbf{R}(v') = \mathbf{S}(v'')$ で、かつ v'' が v に直接データ従属しているような 2 つの節点 v' と v'' が存在するときかつそのときに限り、u は v に**直接通信従属**しているという。 □

通信従属は 2 つのプロセス間の通信によって発生する従属関係である。あるプロセスにある文 v で計算される変数の値がプロセス間の通信によって別のプロセスにある文 u で計算される変数の値に影響を与える可能性がある場合に u は v に通信従属している。

図 2.2で、タスク PV にあるエントリ Add 中の 28 で使用されている変数 X を例にとる。この変数 X はエントリ Add の入力引数であり、その値はこのエントリを環境タスクが 59 で呼び出した際に変数 Value から渡されたものある。変数 Value の値は 57 で定義されている。従って、 28 は 57 に通信従属している。

直接データ従属と直接通信従属の違いは、データ従属はプロセス内部の文の間に存在する従属性であり、プロセス間の通信チャネルとは無関係だが、通信従属はプロセス間に存在する従属性であり、プロセス間の通信チャネルに関係があるという点である。

以上で並行プログラムにおける基本的な 5 つの従属性を定義した。これら 5 つの従属性を枝が分類された有向グラフとして表現することにより、プログラムの従属性を明示的に表現することができる。これをプロセス従属ネットと呼ぶ。

定義 2.3.6 並行プログラムの**プロセス従属ネット** (以下 PDN) は枝が分類された有向グラフ $(\mathbf{V}, \mathbf{Con}, \mathbf{Sel}, \mathbf{Dat}, \mathbf{Syn}, \mathbf{Com})$ である。ここで、 **V** はそのプログラムの制御フローネットにおける節点の集合、 **Con** は制御従属枝の集合、すなわち任意の (u, v) について u が v に直接弱く制御従属しているときかつそのときに限り $(u, v) \in \mathbf{Con}$、 **Sel** は選択従属枝の集合、すなわち任意の (u, v) について u が v に直接選択従属しているときかつそのときに限り $(u, v) \in \mathbf{Sel}$、 **Dat** はデータ従属枝の集合、すなわち任意の (u, v) について u が v に直接データ従属しているときかつそのときに限り $(u, v) \in \mathbf{Dat}$、 **Syn** は同期従属枝の集合、すなわち任意の (u, v) について u が v に直接同期従属しているときかつそのときに限り $(u, v) \in \mathbf{Syn}$、 **Com** は通信従属枝の集合、すなわち任意の (u, v) について u が v に直接通信従属しているときかつそのときに限り $(u, v) \in \mathbf{Com}$ である。 □

従来の逐次プログラムに対する PDG は、 PDN から並行プログラムに特有の選択従属性、同期従属性、および通信従属性を除いた特殊な場合と考えられる。図 2.3に、図 2.1の Ada プログラムの PDN を示す。図 2.2 と同様、各節点内の数字は図 2.1のプログラムにおける行番号に対応している。 5 種類の異なる枝で 5 種類の従属関係を表している。

また、 PDN の各従属性を表す有向枝の向きを全て逆向きにしたグラフを**プロセス支配ネット** (以下 PIN) と呼ぶ。 PIN の各有向枝はある節点がどの節点の実行に直接影響を与えるか (支配しているか) を示す。 PIN を用いることにより、ある文の実行がプログラムのどの範囲に影響を与えるかを知ることができる。これは特に保守などの際に有効な情報を提供する。

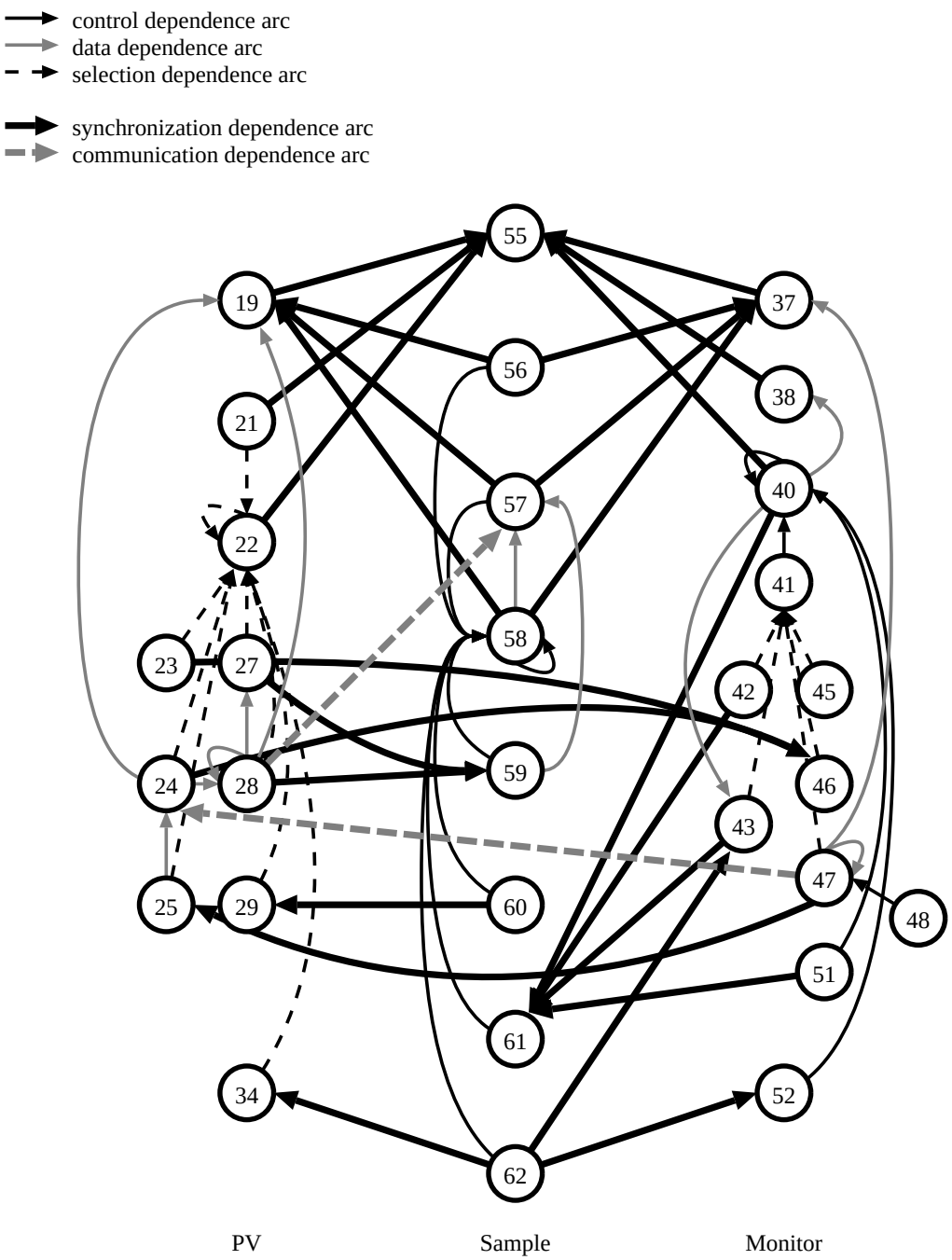

図 2.3: 図 2.1 のサンプルプログラムに関する PDN

2.4　CFN、DUN、および PDN の利用

CFN、DUN、および PDN/PIN は、並行プログラムの表現モデルとして、多くの局面で利用できる可能性がある。以下の各節でそれらの概略を述べる。

2.4.1　プログラムスライス

PDN や PIN の最も直接的な利用として、プログラムスライスの生成がある。 PDN や PIN を利用すれば、並行プログラムのスライスを容易に作成できる。プログラムスライスは、ソフトウェア開発においてさまざまな応用が考えられ研究されており、利用価値が高い [1][17][28][39][42]。

プログラムスライスには大きく分けて静的スライスと動的スライスの 2 種類がある。静的スライスとは、対象プログラム中の注目したい文 s とそこで使われている変数の集合 V について、 s の処理の開始や終了に影響を与える可能性のある文、および、 V に影響を与える可能性のある文のみを抜き出したものである。動的スライスとは、同じく対象プログラム中の注目したい文 s とそこで使われている変数の集合 V について、プログラムのある実行履歴 H に対し、その実行で実際に s の処理の開始や終了、また V に影響を与えた文のみを抜き出したものである。

PDN は対象プログラムのさまざまなプログラム従属性を明示的に表現している。従って、一旦対象プログラムを変換して PDN で表現できれば、静的スライスはネットの到達性問題として容易に計算することができる。同様に、動的スライスは、対象プログラムの実行履歴と PDN を組み合わせることにより、やはりネットの到達性問題に帰結される。 PDN の統一表現は言語に依存しないため、 PDN からプログラムスライスを生成する手法はプログラミング言語に依存しない形で構築することができる。

プログラムの前方スライスは、対象プログラム中のある文の実行に影響を受ける可能性のある文のみをプログラム全体から抜き出したものである。前方スライスに対して、通常のスライスを後方スライスと呼ぶ場合がある。 PIN は対象プログラム中のさまざまな影響関係を明示的に表現している。従って、一旦対象プログラムを変換して PIN で表現できれば、静的前方スライスはネットの到達性問題として容易に計算することができる。同様に、動的前方スライスは、対象プログラムの実行履歴と PIN を組み合わせることにより、やはりネットの到達性問題に帰結される。

2.4.2　最適化・並列化

逐次プログラムの最適化の際には、CFG、DUG が用いられる。同様に、並行プログラムの最適化の際には、CFN、DUN を利用することができる。逐次プログラムの最適化の際に PDG を利用すると効果的に最適化ができることが知られており [19]、PDN や PIN を用いることによりこれを並行プログラムに適用することができる。

PDG は元来、逐次プログラムを自動的に並列化するために考案されたモデルである。逐次プログラム中の文間の従属関係を明らかにすることにより、従属関係のない文を並列的に実行するようにプログラムを変換し、並列化することができる。これを並行プログラムに適用することにより、PDN を用いて並行プログラムをさらに並列化することができる。

2.4.3　理解

プログラムを理解する際には、注目している行がどういう経緯で実行され、これからの実行にどのような影響を与えていくかを把握する必要がある。これはそのプログラムの中に存在するプログラム従属性の把握と考えることができる。プログラマはプログラムを読む際に暗黙的にそれを行っている。しかし、手続き型プログラムの場合、制御の流れは比較的明確にプログラムから読み取れるが、データの流れは制御の流れの中に埋もれていて、注意深く追っていかないと見落しや勘違いを招きがちである。また、並行プログラムでは、複数のプロセス間にある同期や通信を考慮に入れなければならず、従属性の把握はさらに困難になる。

対象プログラムの PDN を自動的に生成し、プログラマがプログラムを読む際に参考にできるように提示することにより、プログラマはプログラムの文間の従属関係を探す作業から解放され、プログラムの振舞いを理解することに集中できる。例えば、注目している文に対し、その文が直接従属している文と、直接支配している文とが別の画面に表示されるようにするだけでも、プログラム理解の大きな助けになると考えられる。また、PDN を視覚化することによって、大まかなプログラム構成の把握や、予期せぬ従属性の発見、あるはずの従属性の欠落などを目で見て確認するといったことができるだろう。

2.4.4　複雑さ計測

PDN と PIN は並行プログラムにおけるプロセス内の制御のデータの流れの性質、プロセス間の同期と通信の性質を表わしていることから、これを用いることにより並行プログラムの複雑さをさまざまな観点から形式的に定義し、また容易に計測することができる [15][43]。たとえば、各従属性の数、並行性に関する従属性の数、ある文に特定の種類の従属性に関して従属している文の数、なんらかの形で他の文に従属している文の数、ある 2 つのプロセスの間で互いに従属関係にある文の数、などである。これらの数値とプログラム全体の文の数や、実行経路中の文の数を比較することにより、並行プログラムに関するより深い複雑さの定義ができる。また、ある並行プログラムに存在する逐次的な従属性と並行的な従属性の割り合いから、そのプログラムがどのくらい並列的であるかといったことを計測することも考えられる。

2.4.5　テスト

CFN、DUN、PDN および PIN にもとづいて、構造テストにおけるさまざまなテストケース評価基準を定義でき、また計算することができる。CFN は並行プログラム中の制御の流れを表していることから、文被覆テスト基準、分岐被覆テスト基準といった、プログラムの文、および実行列に関する評価に用いることができる。DUN は CFN の情報に加えて変数の定義と使用、プロセス間の同期と通信に関する情報を持っているため、定義使用被覆テスト基準といったデータの流れに関する評価に用いることができる。また、相互作用被覆テスト基準のような、プロセス間の相互作用に関する評価に用いることができる。PDN と PIN は並行プログラム中の各種従属関係を表していることから、従属性被覆テスト基準に用いることができる。

2.4.6　デバッグ

プログラムをデバッグする際に、プログラマは誤りが検出された文から遡ってその文の実行に影響を与える文を探し、バグの位置を特定する。PDN は並行プログラムの各文間に存在する従属関係を明示的に示しているため、これを用いることにより、誤りが検出された文をもとにバグが存在する可能性のある文だけを追跡することができる。例えば、C 言語について、プログラムスライスを用いたデバッグ支援ツールに関する研究がなされている [2]。PDN と PIN を用いたプログラムスライス生成ツールを開発することによって、並行プログラムに対する同

様のツールを用意することができる。

2.4.7　保守

ソフトウェアの保守を困難で非効率にしている大きな理由の一つとして、プログラムの一部を変更したときにその変更がプログラム全体のどこまで影響を与えるかを知るのが難しいことがあげられる。並行プログラムの場合、あるプロセスのある文に加えた変更が、どのプロセスのどの文に影響を与えるか、また逆に、その変更した文に影響を与えるのはどのプロセスのどの文かを知る必要がある。PDN および PIN を直接利用したり、これらのネットから求められたプログラムのスライスを用いることにより、変更の影響範囲を知ることが容易になる。

また、Gallagher と Lyle によって、プログラムスライスを用いた逐次プログラムの保守法が提案されている [20]。分解スライスは、逐次プログラムの保守に用いるために提案された一種のスライスである。分解スライスは、対象プログラム中のある特定の変数に対する計算を全て含むプログラムスライス (またはその集合) である。分解スライスを用いることにより、対象プログラムからある特定の変数に関係のある部分だけを分離することができる。この分離された部分をある規則に従って編集し、この部分の動作が正しいかどうかをテストすれば、編集後の部分をもとのプログラムに統合した際にプログラム全体が正しく動作することを保証できる。通常、保守の際にはプログラムの一部分を取り出して改修しその部分だけでのテストを行った後、取り出した部分をプログラム全体に戻し、これに対して再テストを行う必要がある。この変更後のプログラム全体に対する再テストを後退テストと呼ぶが、分解スライスを利用した保守法を用いることにより、この後退テストが必要なくなる。後退テストの削減はソフトウェア保守のコストから見て効果的である。プログラムの分解スライスは PDN から生成することが可能であるため、 Gallagher と Lyle の手法を逐次プログラムのみならず並行プログラムにも適用することができる。

2.5　CFN、DUN、および PDN を利用するための課題

CFN、DUN、および PDN を実際に逐次・並行プログラムの開発支援に利用するためには、解決しなければならない課題がある。この節ではこれらを列挙する。

CFN、DUN、および PDN の自動生成

CFN、DUN、および PDN を開発支援に利用するためには、まず対象プログラムをこれらのモデルに自動変換する必要がある。まず、対象プログラムから CFN/DUN を生成するためには、そのプログラムを記述しているプログラミング言語の言語規則に従ってプログラムを字句・構文解析し、制御の流れと変数の定義・使用の状況、プロセス間の同期・通信の状態を抽出し、CFN/DUN を生成する手法を開発する必要がある。次に、CFN/DUN にもとづいて、各節点間に存在する各種従属性を解析するアルゴリズムを考案し、これを実装して PDN 生成ツールを開発する必要がある。

CFN、DUN、および PDN の統一的表現形式の考案

CFN、DUN、および PDN は、開発支援環境下にあるさまざまなツールで共有され、利用される。従って、複数のプログラムで共有でき、これらのモデルを適切に表現できる表現形式を考案する必要がある。方向性としては、可読文字を用いたテキストによる表現と、内部表現をバイナリで出力した表現が考えられる。テキスト形式には高い可搬性、可読性、加工の容易さといった利点と、メモリ効率の低さ、内部表現への変換のオーバーヘッドといった欠点がある。バイナリ形式はテキスト形式と逆の利点・欠点がある。

手続き呼び出しへの対応

現在、CFN、DUN、および PDN には、手続き呼び出しの表現が定義されていない。実用規模のプログラム開発支援にこれらのモデルを利用するためには、手続き呼び出しへの対応が必要不可欠である。システム従属グラフ [22] のような逐次プログラムの手続き間スライス等に用いられるモデルを参考に、PDN を拡張する必要がある。手続き内部に他のプロセスとの同期・通信が存在しなければ、システム従属グラフの概念をそのまま PDN に適用することができる。しかし、手続き内部に他のプロセスとの同期・通信がある場合にはそのままシステム従属グラフを適用することができないため、新たな枠組を考案する必要がある。

ポインタ、配列への対応

手続きへの対応と同様に重要な課題として、ポインタや配列を含むプログラムへの対応が挙げられる。最もおおざっぱな対応は、同じ型のポインタは全て同じとして扱い、配列も添字に関係なく一つの変数として扱うというやりかたである。しかし、実際には、ポインタの指している場所や配列の添字に起因するバグこそが発見や原因追及の難しいバグであることから、プログラムのデバッグ等を行う際にはこれらのバグを追及したいという要求が強い。これらへの対応は、逐次プログラムで用いられている技法を応用することで対応できる部分が多い。

これらのモデルを利用した支援ツールの作成

CFN、DUN、および PDN にはソフトウェア開発支援の際にさまざまな応用があることを述べた。しかし、実際に支援を行うツールには、プログラムスライス生成ツールや複雑さ計測ツールといった比較的実装が容易なものから、バグの自動発見支援ツールや保守支援ツールといったさらに研究が必要なものまでさまざまなものが存在する。それぞれのツールの開発における問題点を明らかにし、研究を進める必要がある。

支援ツール群の統合とインターフェイスの統一

本研究の最終目標は、逐次・並行プログラムの統合的な開発支援環境の構築であるから、統一的プログラム表現を用いた各種開発支援ツールを開発するだけでなく、これらを統合することによりソフトウェアの開発を一貫して支援する環境に仕上げる必要がある。このためには、各ツール間の相互インターフェイスの設計や、ユーザインターフェイスの設計等さまざまな課題がある。

第 3 章

並行プログラムからの CFN、 DUN および PDN の自動生成

この章では、第2章で述べた CFN、 DUN、および PDN を対象プログラムから自動生成する手法、およびこれを実装したツールについて述べる。

3.1　CFN、 DUN、および PDN の自動生成

本研究では、 CFN、 DUN、および PDN の利用のために、 C、 Pascal、および Ada で書かれたプログラムから CFN/DUN を生成するツールと、 DUN から PDN を生成するツールを開発した。 CFN、 DUN、および PDN は、特定の言語仕様に依存しない形で定義されているため、共通化できる部分はできるだけまとめ、ツールの実装者が新しい言語に対応する際の負担を軽減する方向で研究を進めた。結果として、ソースリストから CFN/DUN を生成する部分はプログラミング言語依存、 CFN/DUN から PDN を生成する部分は言語に依存しない形になっている。

この章ではこのうち並行プログラミング言語 Ada を例に取り、 Ada 並行プログラムから PDN を自動的に生成する手法について詳しく述べる。次いでこれを実装した PDN 生成ツールについて述べる。 Ada 並行プログラムでは、プロセスにあたる並行実行単位を一般にタスクと呼ぶため、この章ではこれ以降「プロセス」のかわりに「タスク」という用語を用いる。また、同様に PDN をタスク従属ネット (TDN) と表記する。

3.1.1　TDN の生成手法

TDN 生成の手順は大きく次の 4 つの段階からなる [26]。

1. 対象プログラムのソースリストを解析し、制御の流れ、変数の定義と使用、およびタスク間の相互作用の情報を抽出し、これらの情報から DUN を生成する。

2. DUN をもとに、各タスクについて、タスク内の制御従属性、データ従属性、選択従属性を求め、各タスクごとの従属グラフを生成する。

3. タスク間の同期通信の情報から、タスク間の従属関係である同期従属性と通信従属性を求める。

4. 求めた 5 種類の従属性をまとめて TDN を生成する。

以下それぞれについて説明する。 4 番目のステップは自明であるので省略する。

3.1.2　CFN/DUN の生成

対象プログラムのソースリストを字句・構文解析することにより、制御の流れ、変数への値の代入と変数の値の使用状況、またあるタスクから別のタスクへのエントリ呼び出し文の場所や、 `accept` 文の開始と終了の場所といった情報を抽出する。また、タスクの親子関係の情報と、親子関係によって待機状態の発生する文の情報を収集する。これらの情報によって CFN/DUN が生成できる。

Ada プログラムの各構文と DUN との対応を全て列挙することは紙面の都合から不可能なのでここでは割愛し、並行処理に関係する部分に関する基本的な戦略だけを述べるに留める。

並行実行枝は、親タスク本体の宣言部 (変数への値の代入などを伴う宣言がなければ実行の直前) から子タスク本体の宣言部へ、また子タスクの終了する文から親タスクの終了する文に接続する。また、子タスクの宣言が終了するまで親タスクは実行を開始しないので、 **S**(子タスクの宣言部の最後)=**R**(親タスクの実行文の最初) の関係がある。

タスク間の同期通信はエントリ呼び出しと受け付けによるランデブで行われる。従って、**S**(エントリ呼び出し文)=**R**(対応する `accept` 文の最初) の関係がある。また、ランデブが終了するまで呼び出し側は待機するという関係から、 **S**(`accept` 文の最後)=**R** (対応するエントリ

呼び出し文の次に実行される文) という関係がある。 Ada では構文上エントリ呼び出し側で呼び出しから戻ってくる文を明示する文がないため、エントリ呼び出し文の次に実行される文を指定している。また、受け渡される引数の使用と定義は、呼び出し側では、呼び出し文で in および in out 引数が使用され、その次の行で out および in out 引数が定義されていると考え、受け付け側では、 accept 文の最初で in および in out 引数が定義され、 accept 文の最後で out および in out 引数が使用されていると考える。

TDN の定義は DUN をもとになされているため [14]、 DUN の情報をもとに TDN を生成することができる。

3.1.3　各タスクごとの従属グラフの生成

定義 2.3.1と 2.3.3で定義した、各タスクの内部に存在する制御従属性とデータ従属性とは、本質的に逐次プログラムにおけるものと同じものである。従って、逐次プログラムのプログラム依存グラフ生成で用いられている手法を使うことができる [3][33]。また定義 2.3.2の選択従属性は、制御従属性と同じアルゴリズムを使って求めることができる。選択従属性と制御従属性との違いは、注目する文が通常の条件分岐文に従属しているか、それとも非決定的な選択文に従属しているかである。これは従属性そのものを求める際には無視することができる。これらのことから、 Ada の各タスクごとの従属グラフの生成は、 Pascal や C の PDG 生成ツールを共用している。制御従属性を求める計算量は $O(n^2)$ であり、データ従属性を求める計算量は $O(n \cdot v)$ である。ここで n は節点数、 v はプログラム中の変数の数である [3][19]。

3.1.4　タスク間の従属関係の抽出

タスク間の相互作用の情報 (並列実行枝および **S**(節点) と **R**(節点) の関係) から、タスク間の従属関係を求める。解析は静的に行うため、同期、通信の可能性がある文間の全てに従属関係があるものとする。すなわち、組となる **S**(節点) と **R**(節点) の全ての組み合せについて、その間に同期従属関係があるとする。

図 3.1に同期従属性を、図 3.2に通信従属性をそれぞれ計算するアルゴリズムを示す。なお、各タスク内のデータ従属性はすでに求められているものとする。

まず同期従属性の求め方について述べる。定義 2.3.4の条件 1) から、並行実行枝と逆向きに

– 入力: DUN – 出力: TDN の同期従属枝部分

```
procedure connectsync(v1,v2);
begin
  v1 から v2 に同期従属枝接続;
  v:=v1;
  while out-degree(v) = 1 loop
    v:=v の先行節点;
    exit when S(v) ≠ ∅ or R(v) ≠ ∅;
    v から v2 に同期従属枝接続;
  end loop;
end connectsync;

begin
  for each (v1, v2) ∈ 並行実行枝 loop
    connectsync(v2,v1);
  end loop;
  for each 節点 v loop
    if S(v) ≠ ∅ then
      for each 節点 u loop
        if S(v) =R(u) then
          connectsync(u,v);
        end if;
      end loop;
    end if;
  end loop;
end;
```

図 3.1: DUN から同期従属性を求めるアルゴリズム

同期従属がある。 Ada では並行実行枝はタスクの親子間に存在するが、どのようなタスクも複数の親を持つことはないので、並行実行枝は t をプログラム中のタスク数とするとたかだか $2 \cdot (t-1)$ 本である。従ってこの部分の計算量は $O(t)$ となる。次に、あるタスクが別のタスクにエントリ呼び出しをしている場合、それらのタスク間には同期従属関係がある。これは定義使用ネット生成時に $\mathbf{S}$ と $\mathbf{R}$ に反映される。これを定義 2.3.4の条件 2) から処理する。図 3.1では全ての節点に対し 2 重ループを組んでいるため節点数 n に対し $O(n^2)$ の処理になるが、実際にはチャネル名からそのチャネルを要素として持つ $\mathbf{S}$(節点) と $\mathbf{R}$(節点) を逆引きできる表をあらかじめ用意することにより線型時間で検索でき、計算量はチャネル数に比例する程度になる。また、定義 2.3.4の条件 3) から、ある節点 v に同期従属している節点に後続する節点で、同じ節点 v に従属すべき節点を求める。

通信従属性は、定義 2.3.5の条件を満たす節点を探し出すことによって求まる。図 3.2では全

– 入力: DUN、各タスクのデータ従属グラフ
– 出力: TDN の通信従属枝部分

```
begin
  for each 節点 v loop
    if S(v) ≠ ∅ then
      for each 節点 u loop
        if S(v) =R(u) then
          u にデータ従属している節点から v がデータ従属している節点
          に通信従属枝接続;
        end if;
      end loop;
    end if;
  end loop;
  同じ大域変数に対する使用と定義の間に通信従属枝を全て接続;
end;
```

図 3.2: DUN から通信従属性を求めるアルゴリズム

節点に対する 2 重ループを組んでいるが、ここも同期従属性の抽出と同様に高速化でき、この部分の計算量もチャネル数に比例する程度になる。

さらに、異なるタスク間で大域変数を定義・参照している場合、そこにデータの流れが発生するため、大域変数について調べる必要がある。解析は静的に行うため、同じ変数の値を定義・使用している文全ての組合せに関して、通信従属性があるものとする。全ての大域変数の値が全てのタスクで定義され、また使用されている場合を想定すると、この部分の計算量は $O(g \cdot t^2)$ となる。ここで g は使用されている大域変数の数である。

3.2　CFN/DUN 生成ツールと TDN 生成ツール

第 3.1.1節の手法にもとづき Ada のための CFN/DUN 生成ツールと、 TDN 生成ツールのプロトタイプを開発した [26]。ツールの構成の概要は図 3.3のとおりである。

CFN/DUN 生成部では対象プログラムを字句・構文解析することにより CFN/DUN を生成する。本ツールでは字句・構文解析に aflex と ayacc を利用した [35][38]。 aflex と ayacc にはこれらで利用可能な Ada の文法規則が付属している。これに CFN/DUN を生成するアクションを付加していくことにより、 CFN/DUN 生成ツールを開発した。このツールにより出力された DUN の例を図 3.4に示す。

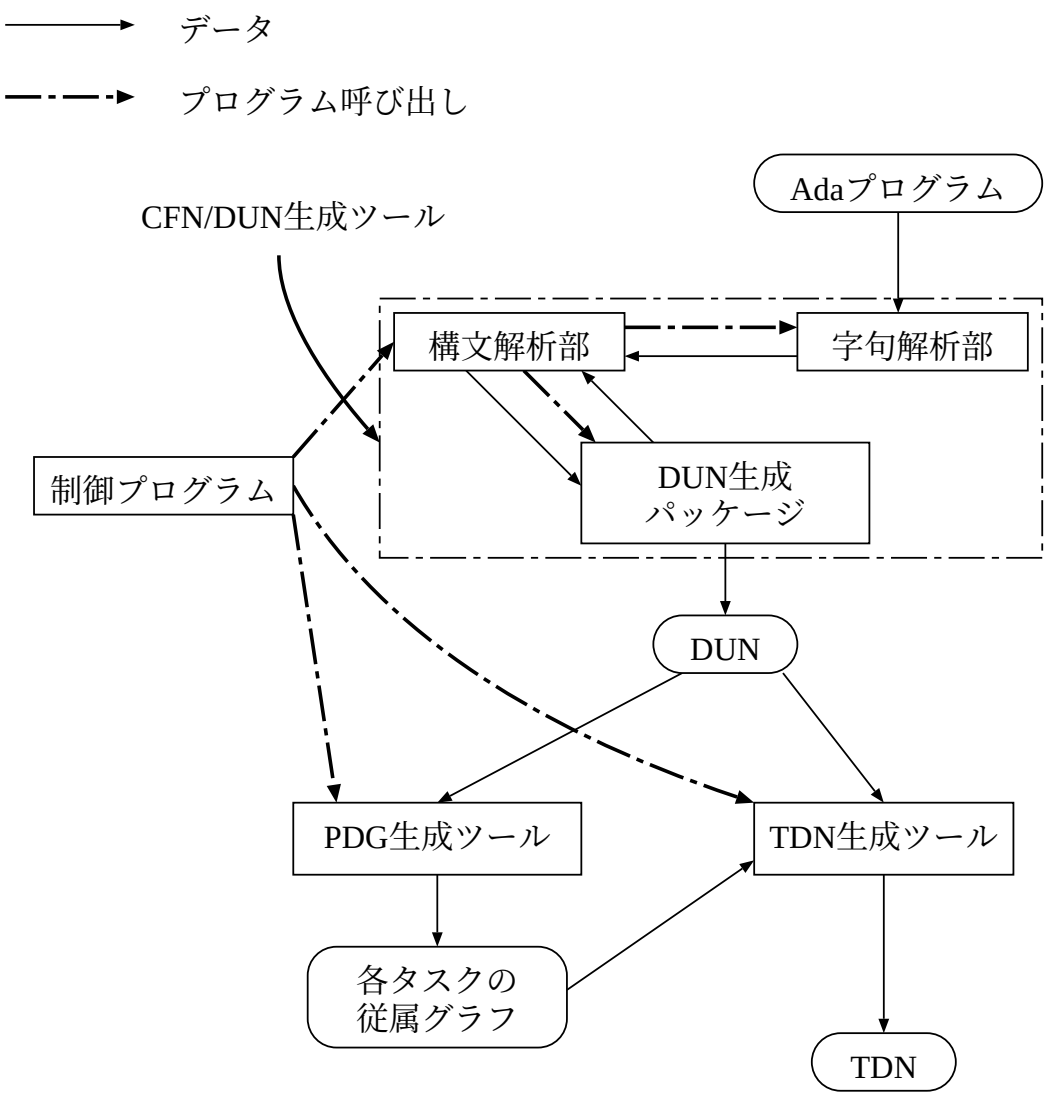

図 3.3: CFN/DUN 生成ツールと TDN 生成ツールの構成図

次に、 3.1.1節で述べた生成手法に従い、 DUN にもとづいて PDG 生成ツールでタスク内の従属関係、 TDN 生成ツールでタスク間の従属関係を解析し、 TDN を生成する。

このツールにより出力された図 2.1のプログラムに対する TDN を図 3.5に示す。この出力自体はそのまま人間が読むのに適しているとは言えないが、 TDN をこのように明示的に自動生成することにより、この出力を他のツールで利用して Ada 並行プログラムの開発支援に活用することができる。 PDG 生成ツール、 TDN 生成ツールは言語に依存しない DUN の情報のみを利用しているため、新しい言語の PDN を生成したい場合にはその言語用の CFN/DUN 生成ツールを開発するだけでよい。

```
 1: <line> 7 <connect> 27 <def> MAIN.Value-1
 2: <line> 18 <connect> 3
 3: <line> 19 <connect> 4 <def> MAIN.PV.V-2 <send> MAIN
 4: <line> 21 <connect> 5
 5: <line> 22 <s-connect> 6 9 12
 6: <line> 23 <connect> 7 <receive> MAIN.PV.Read
 7: <line> 24 <connect> 8 <def> MAIN.PV.X-3 <use> MAIN.PV.V-2
 8: <line> 25 <connect> 13 <use> MAIN.PV.X-3 <send> MAIN.PV.Read!END
 9: <line> 27 <connect> 10 <def> MAIN.PV.X-4 <receive> MAIN.PV.Add
 10: <line> 28 <connect> 11 <def> MAIN.PV.V-2 <use> MAIN.PV.X-4 MAIN.PV.V-2
 11: <line> 29 <connect> 13 <send> MAIN.PV.Add!END
 12: <line> 31 <connect> 14
 13: <line> 33 <connect> 4
 14: <line> 34 <p-connect> 34
 15: <line> 36 <connect> 16
 16: <line> 37 <connect> 17 <def> MAIN.Monitor.V-5 <send> MAIN
 17: <line> 38 <connect> 18 <def> MAIN.Monitor.Finish-6
 18: <line> 40 <connect> 19 26 <use> MAIN.Monitor.Finish-6
 19: <line> 41 <s-connect> 20 22
 20: <line> 42 <connect> 21 <receive> MAIN.Monitor.Quit
 21: <line> 43 <connect> 25 <def> MAIN.Monitor.Finish-6 <send> MAIN.Monitor.Qui
t!END
 22: <line> 46 <connect> 23 <send> MAIN.PV.Read
 23: <line> 47 <connect> 24 25 <def> MAIN.Monitor.V-5 <use> MAIN.Monitor.V-5 <r
eceive> MAIN.PV.Read!END
 24: <line> 48 <connect> 25
 25: <line> 51 <connect> 18
 26: <line> 52 <p-connect> 34
 27: <line> 55 <connect> 28 <p-connect> 16 3
 28: <line> 56 <connect> 29 <receive> MAIN
 29: <line> 57 <connect> 30 <def> MAIN.Value-1
 30: <line> 58 <connect> 31 33 <use> MAIN.Value-1
 31: <line> 59 <connect> 32 <use> MAIN.Value-1 <send> MAIN.PV.Add
 32: <line> 60 <connect> 28 <receive> MAIN.PV.Add!END
 33: <line> 61 <connect> 34 <send> MAIN.Monitor.Quit
 34: <line> 62 <receive> MAIN.Monitor.Quit!END
```

図 3.4: CFN/DUN 生成ツールの出力例

Ada でのタスクの生成には、プログラムのソースリスト中に静的にタスクの生成を宣言するという方法と、ソースリスト中にはタスクを型として宣言し、割当子によって (場合によっては同じ型で複数の) タスクを実行時に動的に生成するという方法がある。本ツールは前者を含む Ada 並行プログラムのソースリストから TDN を生成し出力することができるが、静的に解析を行っているため、動的に生成される後者を含む場合には対応できない。

3.3　成果と今後の課題

本研究では、統一的プログラム表現の基礎となる CFN、 DUN、および PDN の生成ツール開発を行った。具体的には、 Ada、 C、 Pascal プログラムから CFN/DUN を生成するツール

```
1: <line> 7
2: <line> 18
3: <line> 19 <sync> 27
4: <line> 21 <sele> 5 <sync> 27
5: <line> 22 <sele> 5 <sync> 27
6: <line> 23 <sele> 5 <sync> 22
7: <line> 24 <data> 3 10 <sele> 5 <sync> 22
8: <line> 25 <data> 7 <sele> 5
9: <line> 27 <sele> 5 <sync> 31
10: <line> 28 <data> 3 10 9 <sele> 5 <sync> 31 <comm> 29
11: <line> 29 <sele> 5
12: <line> 31
13: <line> 33 <sele> 5
14: <line> 34
15: <line> 36
16: <line> 37 <sync> 27
17: <line> 38 <sync> 27
18: <line> 40 <control> 18 <data> 21 17 <sync> 27 33
19: <line> 41 <control> 18
20: <line> 42 <sele> 19 <sync> 33
21: <line> 43 <sele> 19 <sync> 33
22: <line> 46 <sele> 19
23: <line> 47 <data> 16 23 <sele> 19 <sync> 8 <comm> 7
24: <line> 48 <control> 23
25: <line> 51 <control> 18 <sync> 33
26: <line> 52
27: <line> 55
28: <line> 56 <control> 30 <sync> 3 16
29: <line> 57 <control> 30 <sync> 3 16
30: <line> 58 <control> 30 <data> 29 <sync> 3 16
31: <line> 59 <control> 30 <data> 29
32: <line> 60 <control> 30 <sync> 11
33: <line> 61 <control> 30
34: <line> 62 <control> 30 <sync> 26 14 20
```

図 3.5: TDN 生成ツールの出力例

と、統一的な CFN/DUN から PDN を生成するアルゴリズムの考案およびその実装を行った。このツール群は、言語に依存する部分と依存しない部分を分割して実装してあり、今後新しいプログラミング言語で書かれたプログラムの従属性解析を行いたい場合には、 CFN/DUN 生成ツールの部分だけを実装するだけでよいという特徴がある。

本研究で開発した CFN/DUN 生成ツールおよび TDN 生成ツールには、以下のような研究課題が残されている。

本ツールを実用規模のプログラムに適用するためには、手続き呼び出しの処理ができなければならない。現在のところ、これらのツールのもとになっている CFN、 DUN、および PDN の定義そのものに手続き呼び出しの概念がないため、本ツールも手続き呼び出しに対応できていない。まずもとになっているモデルそのものを改良していく必要がある。また、逐次プログラムの最適化、並列化、従属性解析等で用いられているポインタ、エイリアス、配列の添字解

析の手法を導入することにより、これらに対応していく必要がある。

また、 Ada95、 Occam2 等、より多くのプログラミング言語で書かれたプログラムに対する CFN/DUN 生成ツールの開発を行うとともに、 CFN、 DUN、および PDN がこのようなさまざまな手続き型プログラムを表現するのに十分かどうかの検証とモデルの改良を行う必要がある。

第 4 章

開発支援環境の構築

この章では、第2章で述べたCFN、DUN、およびPDNにもとづく統合的なソフトウェア開発支援環境の構築について述べる。

4.1　統一的プログラム表現を用いたソフトウェア開発支援

ソフトウェアの開発や保守の際には、プログラマは対象プログラムの動作を把握するためにプログラムを注意深く読む必要がある。例えば、「この変数の値はどこで定義されているのか」「この変数の値はどこで使われているのか」「この文はどういう条件が満される時に実行されるのか」といった情報は、プログラムの動作を把握する際に不可欠である。しかし、これらの情報はプログラム中に暗黙的に含まれており、簡単に読み取ることができない場合も多い。さらに、並行プログラムには複数の制御の流れとデータの流れが存在し、これらがプログラム中で非決定的に相互作用しているため、並行プログラムを理解するのはさらに困難である。CFN、DUN、およびPDNはこのような情報を明示的に表現している。従って、CFN、DUN、およびPDNを対象プログラムの抽象表現モデルとして用いることにより、プログラマは対象プログラムを容易に理解できるようになり、ソフトウェアの開発・保守を効果的に行うことができる。

一般に、大規模分散処理システムは多くの部品からなっていることが多く、また各部品はいくつかの異なるプログラミング言語で書かれている場合もある。通常、各プログラミング言語はそれぞれに固有の開発環境を持っている。各環境はそれぞれ異なるユーザインターフェイス

を持っている場合が多く、また複数の異なる環境を頻繁に行き来するのは非常に繁雑で手間がかかる。このことから、複数のプログラミング言語で書かれた多様なプログラムを、一つの統合的な環境で扱えるようにすることは有用である。そのためには、プログラミング言語に依存しない、プログラムの統一的抽象表現が必要である。CFN、DUN、および PDN はプログラミング言語に依存していないため、一旦対象プログラムをこれらの表現に変換してしまえば、これを言語に依存しない形で取り扱うことができる。

第 2.4 節で述べたように、CFN、DUN、および PDN はソフトウェア開発活動のさまざまな場面で利用できるため、これらのモデルにもとづいてさまざまなソフトウェア開発支援ツールを開発することができ、またこれらのツールはほとんどが言語に依存しない形で実装できる。このことにより、今後本環境を新しい言語に対応したい場合にも、その言語用の CFN/DUN 生成ツールを実装するだけで、言語に依存しないツール群に関しては即座に利用可能となる。

以上の議論により、CFN、DUN、および PDN はソフトウェアの開発・保守におけるさまざまな活動を支援する際に利用できる、対象プログラムの統一的抽象表現として適切であると考える。

4.2　統合的ソフトウェア開発支援環境

この節では、現在開発中の統合的ソフトウェア開発支援環境の構成について述べる。

4.2.1　統合的開発支援環境の概要

第 4.1節の考察にもとづいて、現在ソフトウェアの開発・保守を支援する統合環境を開発している。統合環境の全体像を図 4.1に示す。

ソースリスト編集 / 表示ツールは統合的支援環境の「顔」であり、統一的なユーザインターフェイスをユーザに提供する。通常ユーザはこのツールを通して統合環境下の他のツールを利用する。このツールはソースリスト、CFN、DUN、PDN 等の対象プログラムに関する情報を保持し、ユーザからのコマンドを受けて統合環境下のさまざまなツールを起動し、それらとの間でデータの入出力を行った結果を (必要に応じて加工し) ユーザに提示する。

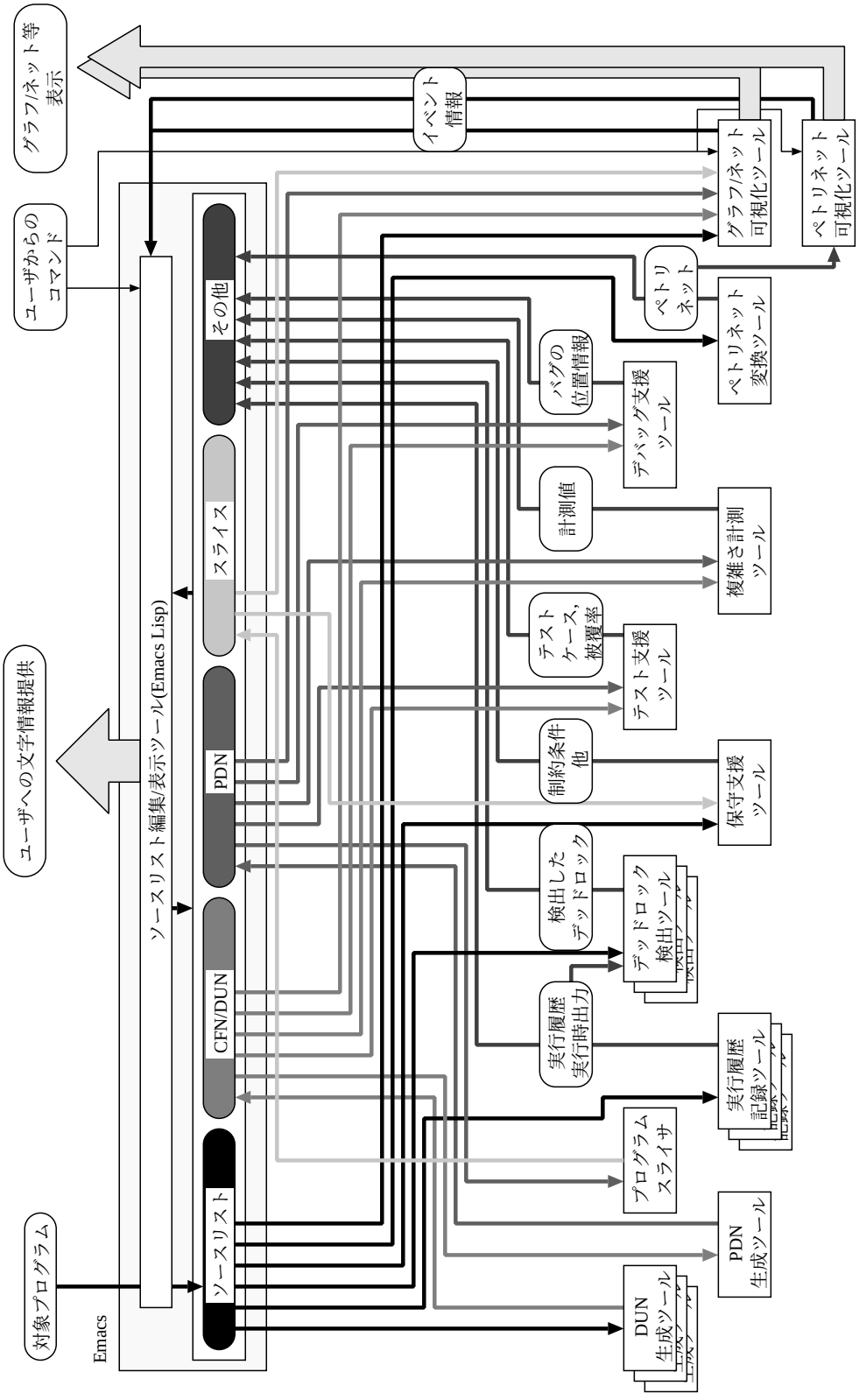

図 4.1: 統合的支援環境の構成

統合環境下の各ツールを環境の統一的なユーザインターフェイスを介さずに独立した形で利用することもできる。必要ならユーザが独自に開発したツールから各種ツールを呼び出し、対象プログラムの解析に活用することができる。

4.2.2　ソースリスト編集 / 表示ツール

UNIX のフィルタ系のコマンド群と同様に、統合環境で用いられるツール群のほとんどは独立したユーザフレンドリなインターフェイスを持たない。ソースリスト編集 / 表示ツールは、その他のツール群に対するユーザフレンドリなインターフェイスとして働く。ユーザ側から見ると、このツールが統合環境そのものに見える。

このインターフェイスは、 Emacs エディタ上に構築している。 Emacs エディタは lisp インタプリタを内蔵しており、これによって機能拡張ができる。 Emacs はエディタとしても優秀であり、また非常に拡張性に富んでいるため、本統合環境のベースとして用いるのに適当だ考えた。ユーザが統合環境の機能を利用する際には、 Emacs 上でいくつかのキーを押し、場合によって Emacs からの質問に答えるだけでよい。

例えば、対象プログラムのスライスを生成する時に、このインターフェイスがない場合とある場合を考えてみる。

このインターフェイスがない場合、ユーザはスライスを生成するのに必要なツールを自分で選び、それぞれに正しい引数を与え、 UNIX のパイプラインを利用するなどして順番に各ツールの結果を次のツールに受け渡して結果を得なければならない。しかも、得られた出力は単なる行番号の列なので、さらにこの結果をもとのソースリストと対応づけする作業が必要である。

このインターフェイスを用いる場合、ユーザは画面に表示されているソースリストの中のスライスを取りたい行にカーソルを合わせ、スライスを生成するコマンドを発行するキーを押せばよい。そのキーによって起動された lisp 関数がスライスを生成するために必要なツールに必要な情報を渡して順に起動する。最終的に得られた行番号のリストから、画面に表示されているソースリストのうちスライスに含まれている行を強調表示する。ソースリスト編集 / 表示ツールの実行例を図 4.2に示す。この図では、図 2.1の Ada プログラムを表示して、カーソル

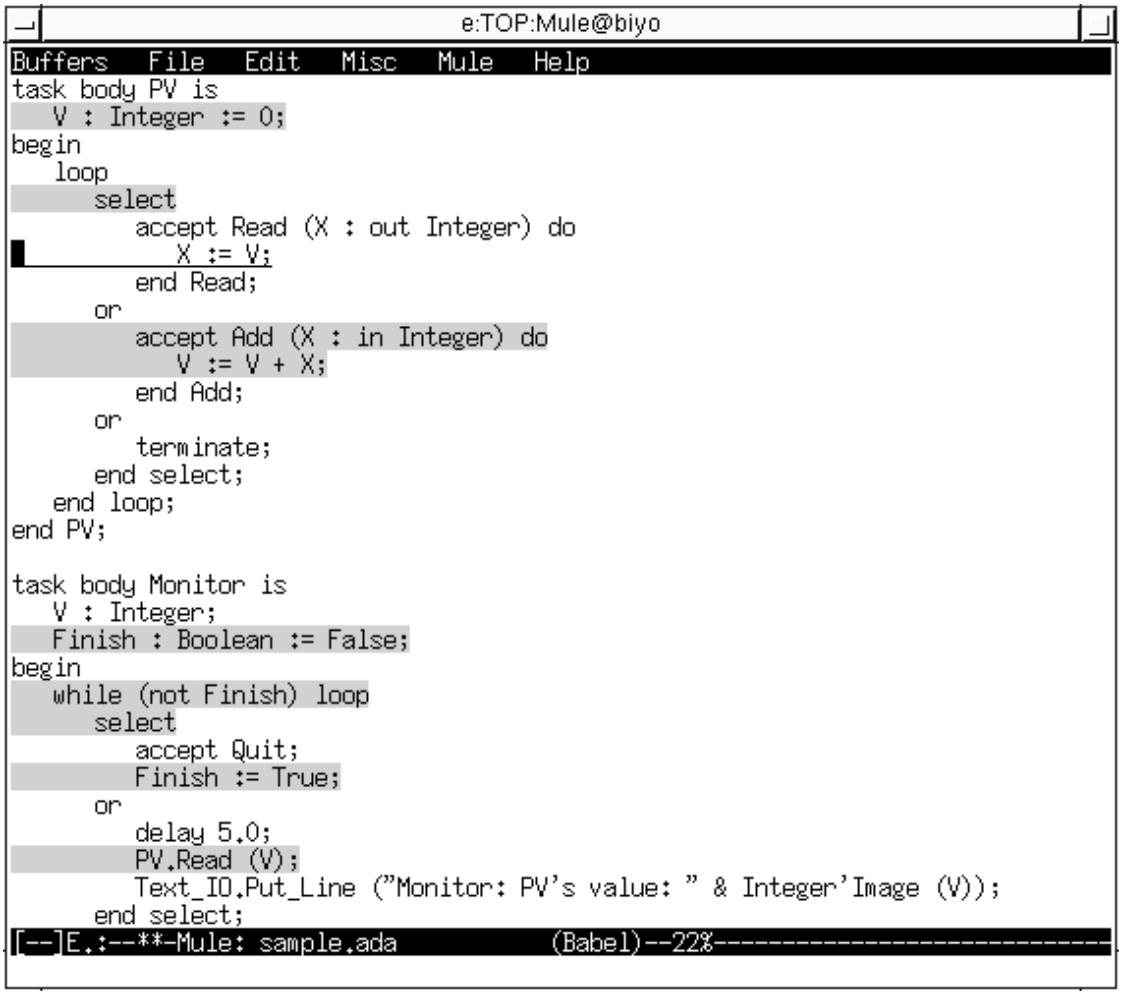

図 4.2: ソースリスト編集 / 表示ツールによるプログラムスライスの表示例

位置からスライスを取っている。

4.2.3　CFN/DUN 生成ツール

この統合環境では、対象となるプログラミング言語それぞれに対して、その言語で書かれたプログラムに対応した CFN/DUN 生成ツールを提供する。図 4.3の上部に描かれている 4 つの箱が、 Ada、 Occam2、 C、および Pascal 用の CFN/DUN 生成ツールを表している。第 3章で述べたように、現在われわれは Pascal、 C、および Ada 用の CFN/DUN 生成ツールのプロトタイプを実装済みである。また現在、 Occam2 用のツールを開発中である。

各 CFN/DUN 生成ツールは、対応するプログラミング言語で書かれたプログラムのソースリストを読み込み、字句・構文解析によって対象プログラム中の決定的および非決定的な制御

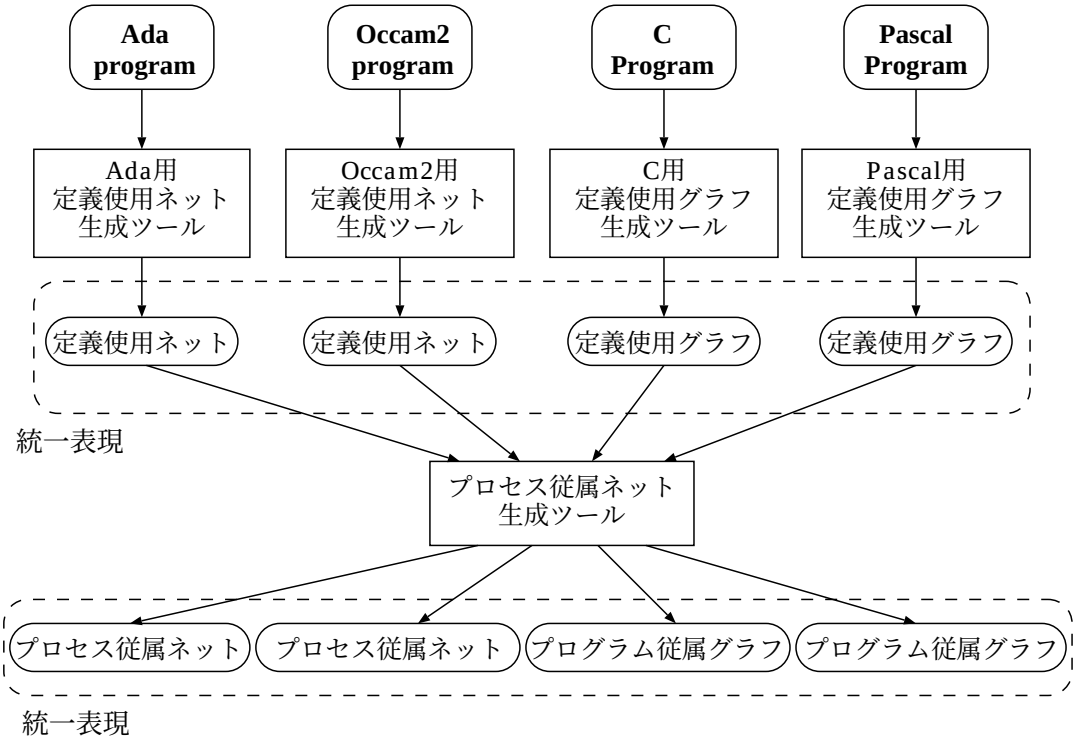

図 4.3: CFN/DUN 生成ツールと PDN 生成ツール

の流れ、変数の定義と使用の状況、プロセス間の相互作用を抽出する。各 CFN/DUN 生成ツールは yacc と lex(またはその同等ツール) を用いて実装されている。ソースリストの字句・構文解析の際、もとのソースリストと CFN/DUN を対応づけるためのシンボルテーブルも同時に作成される。 CFN と DUN はプログラミング言語に依存しない形になっているため、CFN や DUN を用いてもとのソースリストを解析するツールではこのシンボルテーブルが必要になる場合がある。 CFN/DUN 生成ツールは対象プログラムの DUN をテキスト形式でファイルに書き出す。また、シンボルテーブル等の追加情報を別のファイルにやはりテキストファイルで書き出す。この追加情報は図 4.1と図 4.3には表記していない。

ユーザが CFN や DUN の情報を用意に取り扱えるようにするために、 DUN の表現には可読テキストを用いている (第 3章の図 3.4)。この表現はツールや開発プラットフォームと独立である。ユーザが自分で CFN や DUN にもとづくソフトウェア開発・保守支援ツールを開発することも容易である。可読テキストなので、ユーザが直接 DUN の出力を読んで意味を理解することもできる。ソフトウェアが複数のプラットフォームで開発されている場合でも、対象プログラムに関する CFN や DUN の情報をテキスト転送機構によって簡単に共有することがで

きる。このような機構はほとんどのプラットフォームで用意されており、ファイル形式の変換といった面倒な手続きも必要ない。

4.2.4　PDN 生成ツール

PDN 生成ツールは対象とするプログラミング言語に依存しない形で実装され、各種手続き型プログラミング言語で書かれたプログラムのプログラム従属性を解析するのに用いられる。第 3 章で述べたように、われわれはすでに DUN の統一表現にもとづいた PDN 生成ツールのプロトタイプを実装している [26][16]。図 4.3に CFN/DUN 生成ツールと PDN 生成ツールの関係を示す。

4.2.5　プログラムスライス生成ツール

プログラムスライス生成ツールは、対象プログラムのプロセス従属ネットから、指定された文に関する前方・後方スライスを生成する。また、指定された変数に関する分解スライス [20] を生成する。第 4.1節で述べたように、 PDN にもとづくプログラムスライス生成ツールの開発は容易である。

このツールは、対象プログラムの PDN が格納されているファイル名、スライスを生成しはじめる行 (必要なら変数名も)、スライスの種類、スライスを取る際に考慮する従属関係の種類等を引数に指定して起動される。ソースリスト編集 / 表示ツールはこの出力をもとにユーザに対象プログラムのスライスを明示する。

動的スライスを生成する際には対象プログラムの実行履歴が必要であるが、実行モニタはプログラミング言語依存である。われわれはすでに Pascal、 C、および Ada の実行モニタを開発している。これらを用いることにより、動的スライス生成に必要な実行履歴を収集することができる。

4.2.6　グラフ / ネット可視化ツール

このツールは、 CFN、 DUN、 PDN およびプログラムスライスを図として表示する。ソースリスト編集 / 表示ツールがプログラムのソースリストを中心に従属性の追跡表示などを行うのに対して、可視化ツールはグラフ / ネット自体の表示を中心とする。

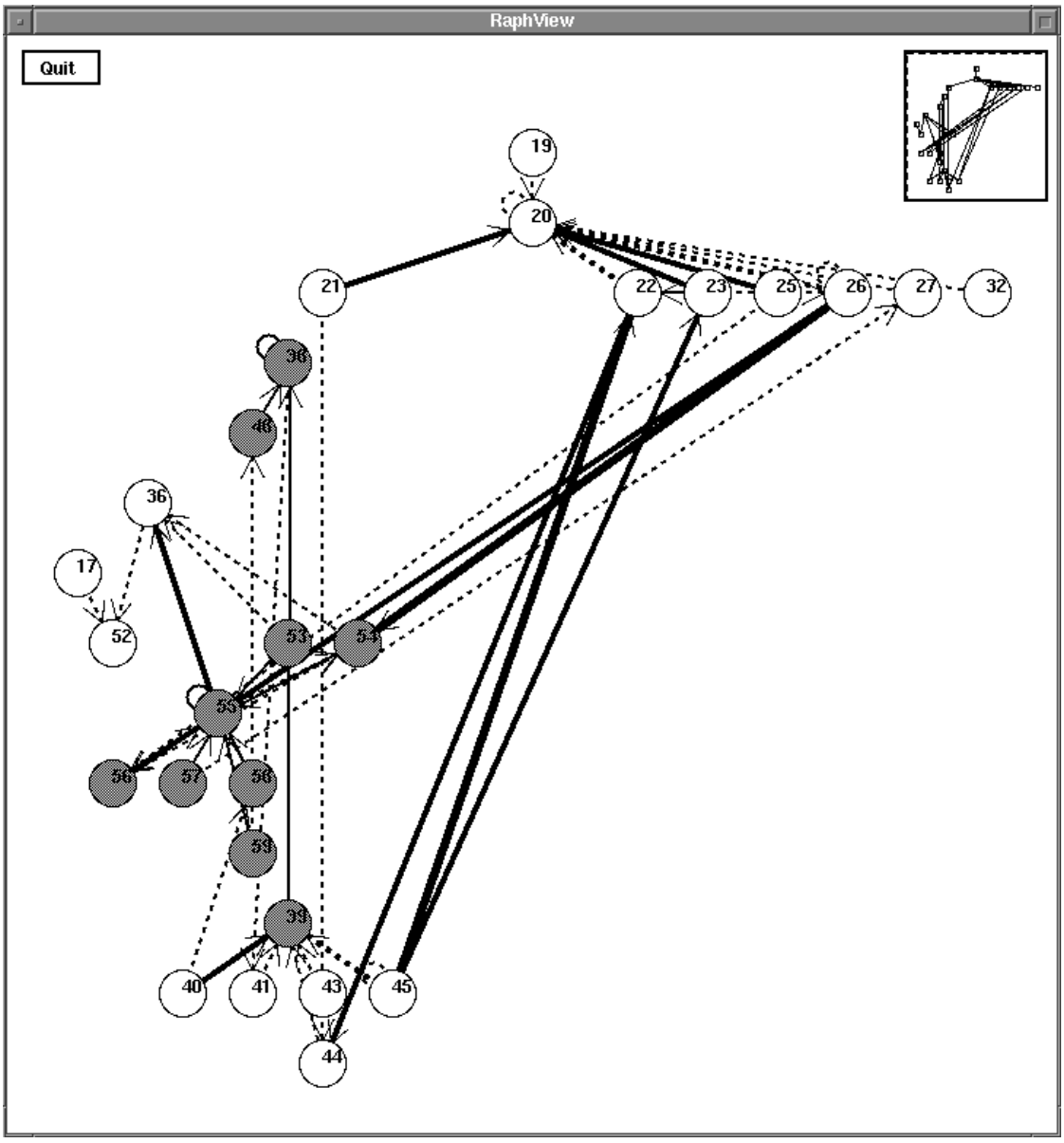

図 4.4: グラフ / ネット表示ツール

このツールはソースリスト編集 / 表示ツールと連携して、「ネット上の任意の点を指定すると対応する文を表示する」、「ソースリストの任意の範囲を指定すると対応するネットを表示する」といった相互の関係を表示することによって、プログラムの構成の把握を支援する。

このツールは CFN、 DUN、または PDN のテキスト形式のファイルを読み込み、あるアルゴリズムで節点と枝の接続関係を解析して配置し、ネットとして画面に表示する。また同時に対象プログラムのソースリストを併用してネットの可読性を増している。このツールによる出力の例を図 4.4に示す。この出力は図 3.5に示したファイルをもとにしている。われわれは現在

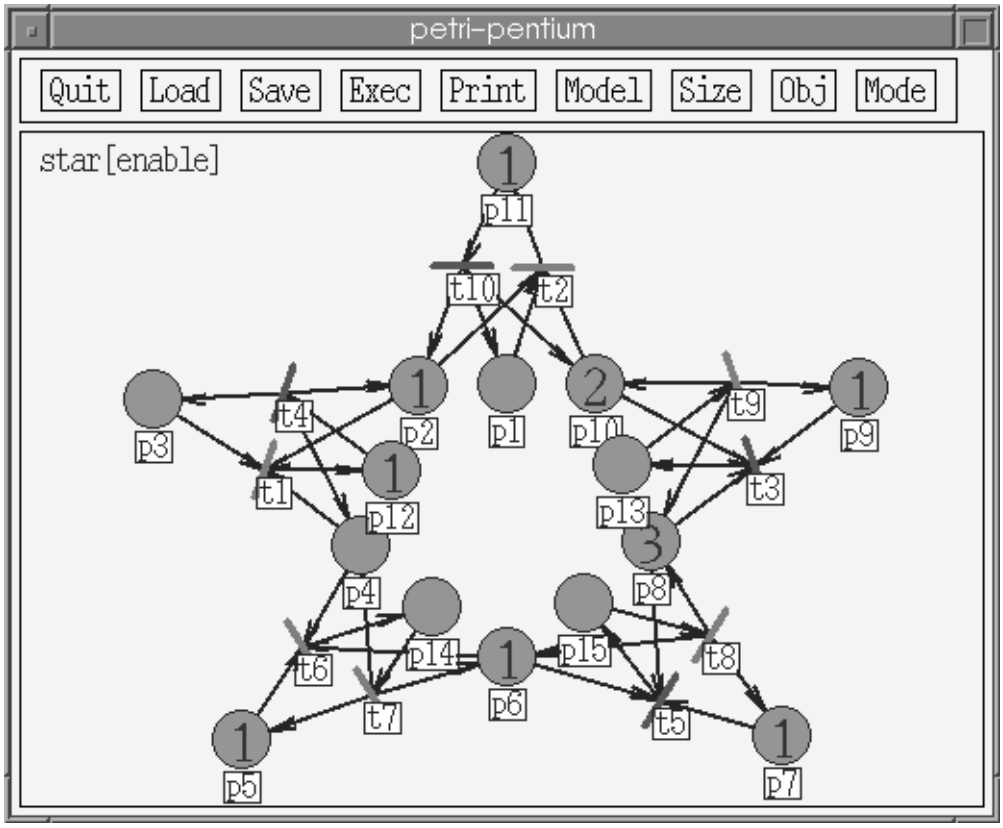

図 4.5: ペトリネット編集・シミュレーションツール

表示されたネットの可読性を増すために節点の配置アルゴリズムを改良している。

図 4.1にあるように、このツールはソースリスト編集 / 表示ツールとは独立のユーザインターフェイスを持っている。ユーザはマウス等で表示を直接操作することができる。従って、ユーザはこのツールを統合環境から独立した状態で利用することもできる。ソースリスト編集 / 表示ツールから起動されている場合には、両ツール間でコマンドのやりとりをすることができる。例えば、ソースリスト編集 / 表示ツールにコマンドを与えることにより可視化ツールの表示が切り換わったり、可視化ツールにコマンドを与えることによりソースリスト編集 / 表示ツールに表示している対象プログラムの表示が切り換わったりする。

4.2.7　ペトリネット編集・シミュレーションツール

ペトリネットは、 C. A. Petri の学位論文において誕生した、同時進行・並列的システムをモデル化する数学的道具の一つである。ペトリネットはシステムの視覚化、シミュレーショ

ン、および数学的解析という三種の応用を同時に兼ね備えたモデルであり、並行プログラムの表現モデルとしても有用である。われわれはペトリネットの視覚化、編集、およびシミュレーションを行うツールを開発しており、これを本環境に統合する予定である [31]。図 4.5にこのツールの表示例を示す。これを Ada プログラムをペトリネットに変換するツールと組み合わせることで、 Ada プログラムの実行をペトリネット上でシミュレートすることができる。

4.2.8　デッドロック検出ツール

デッドロックは並行プログラムにとって複雑でかつ致命的な問題である。デッドロックが発生すると、そのデッドロックに関係するプロセスは動作を停止してしまい、システム全体が正常に動作しなくなってしまう可能性がある。従って、統合的開発支援環境には、並行プログラム中に存在するデッドロックを検出しユーザに報告する機構が用意されるべきである。また可能ならば検出されたデッドロックを解決したり、発生する可能性のあるデッドロックを予測し回避することが望ましい。

われわれはすでに Ada 並行プログラムのデッドロック検出ツールを開発しており [10]、現在これを本環境に統合している。このツールは Ada プログラムの実行モニタを使って、現在実行中の Ada プログラムにデッドロックが起こっているかどうかを調べることができる。同時に、対象プログラムの CFN、 DUN、および PDN を用いて、検出されたデッドロックの原因を解析し、解決するために有用な情報をユーザに与える機構を提供する。

4.2.9　テスト支援ツール

CFN、 DUN、および PDN を用いることにより、テストケースの自動生成やテストケースの評価といったプログラムテストの支援ができる。 PDG を用いた逐次プログラムに対するテストケースの自動生成やテストケースの被覆率計算に関する研究がいくつか行われており、これを PDN を用いた並行プログラムのテストに応用することができる。

われわれは現在従属関係の観点からテストケースの被覆率を計算するツールを開発中である。被覆率はテストケースの質を評価する基準の一つである。このツールは PDN とテストケースの実行履歴を利用してそのテストケースに関する従属関係の被覆率を計算する。

また、われわれは現在対象プログラムに存在する全ての従属関係をテストできるようなテス

トケースの自動生成するツールを開発中である。しかし、現在の PDN の表現には条件分岐文の制御論理式といったテストケースの生成に必要な情報を含んでいないため、 PDN を拡張する必要がある。このような情報はその他のツールにとっても有用な場合がある。

4.2.10　デバッグ支援ツール

プログラムをデバッグする際に対象プログラムのプログラムスライスを用いることにより、プログラム全体からバグが存在する可能性のある場所を抜き出すことができる。この統合環境を用いることにより、ユーザは対象プログラムのデバッグに役だつさまざまなスライスを利用することができる。

しかし、プログラムのスライスは単にバグの候補を示すだけに過ぎず、バグの位置を特定したり、バグの性質のヒントを与えるものではない。ユーザはスライスとソースリストを注意深く調べることにより正しいバグの位置を発見しなければならない。

プログラムをデバッグする際のバグ位置特定、分析、および修正を支援する、より強力な方法とツールを開発するために、われわれは現在、相関論理にもとづいたプログラム中のバグに関する因果関係推論法と、知識ベースにもとづいたプログラムのデバッグ手法を開発している。この時、 CFN、 DUN、および PDN は対象プログラム内の因果関係を表現する基礎として働く。

4.2.11　保守支援ツール

このツールは主に対象プログラムの分解スライスを用いることによりソフトウェアの保守を支援する。ユーザが対象プログラムのある部分を変更しようとする場合に、このツールはプログラムスライス生成ツールにその部分に関する分解スライスを生成させ、これをソースリスト編集 / 表示ツールを通してユーザに提示する。またこのツールはソースリスト編集 / 表示ツールと協力して、ユーザが編集規則を破った場合に警告する。例えば、ユーザが分解スライスの外にある変数を分解スライスの内部に持ち込もうとした場合、エディタがそれを保守支援ツールに知らせ、保守支援ツールは新しい分解スライスを計算しなおす。また、このツールはユーザからの指示によって自動的に分解された各部分を統合してもとのプログラムに再構成する。

また、プログラムの前方スライスと後方スライスはそれぞれある文が影響を与える文の集

合、影響を受ける文の集合を示すことから、これをユーザに提示することによって、プログラム保守の際のコードの書き換え時に起こるプログラム全体への影響範囲を知らせることができる。

4.2.12　複雑さ計測ツール

このツールは CFN、 DUN、および PDN のさまざまな性質を用いてプログラムの複雑さを計測する。われわれはすでに PDN にもとづく計測基準をいくつか提案している [15]。このツールは対象プログラムの CFN、 PDN、および PDN のさまざまな特徴、例えば節点とある種の枝の数の割合等を用いて、そのプログラムの複雑さをさまざまな観点から計測する。

4.3　現在実装されている環境

現在までに実装され、統合されているツール群は次の通りである。

- ソースリスト編集 / 表示ツール
- CFN/DUN 生成ツール (Ada、 C、 Pascal 用)
- PDN 生成ツール
- スライス生成ツール (静的前方 / 後方スライスのみ)
- グラフ / ネット表示ツール

また、実装されているものの、まだ未統合のツール群は次の通りである。

- ペトリネット編集・シミュレーションツール
- デッドロック動的検出ツール (Ada83 用)
- 実行履歴取得ツール (Ada、 C、 Pascal 用)

この節では、実装されているツール群のうち、現在すでに統合されている部分について、実際の利用の様子を画面出力とともに説明する。

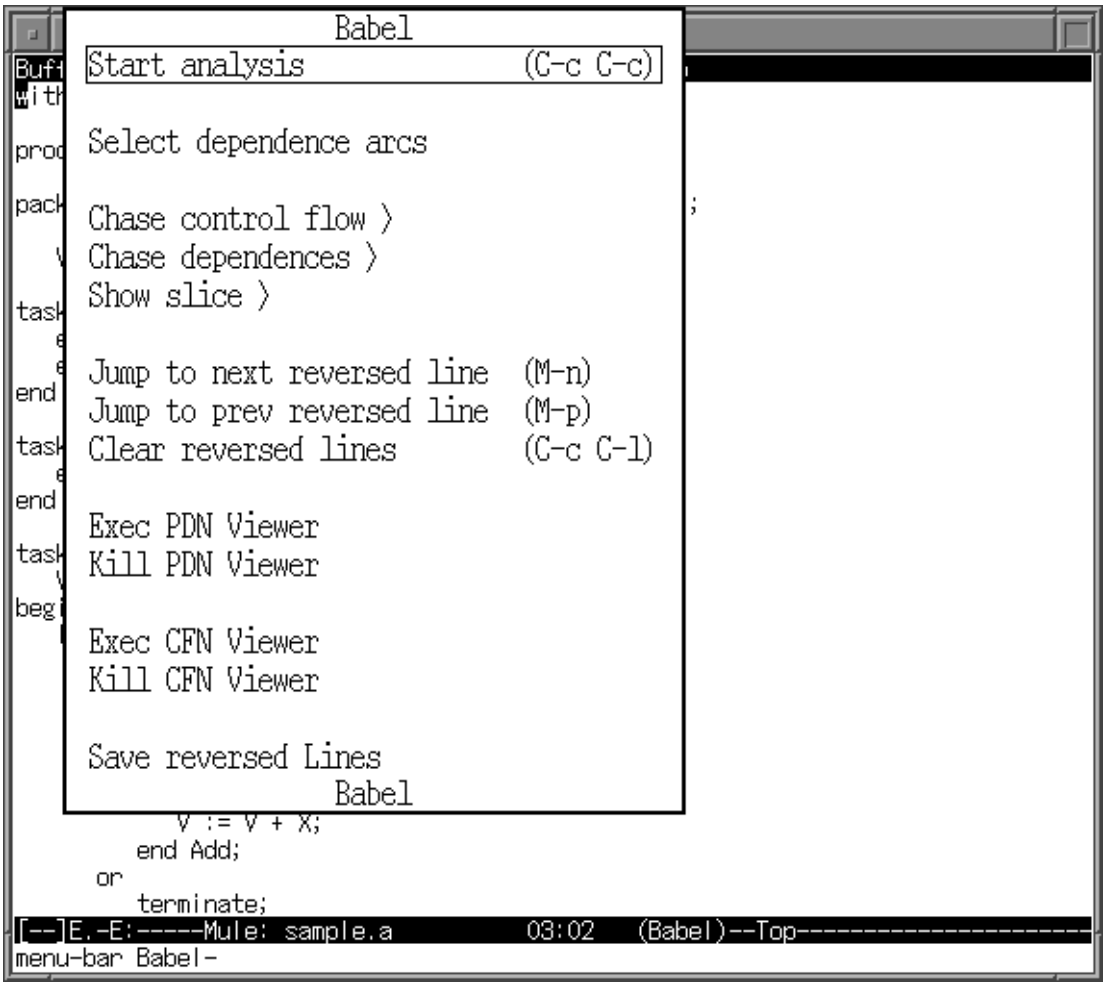

図 4.6: メニュー表示

第 4.2 節で述べたように、本環境は Emacs エディタ上に構築されている。図 4.6に、 Emacs で簡単な Ada プログラムを読み込んで本環境を起動し、環境のメニューを表示している画面を示す。現在この環境で利用可能な機能は全てこのメニューから呼び出すことができるようになっている。また、ユーザは各種機能を好きなようにキーボードに割りあてて簡単に呼び出せるように設定することもできる。メニューの項目の右側の括弧内の表示がそのコマンドに割り当てられているキーを表している。図 4.6は、メニューから対象プログラムの解析を行おうとしている所である。現在はこのようにユーザが明示的に対象プログラムの解析を行って欲しいことを環境に知らせるようになっている。この機能を呼び出すと、 Emacs 上で動作している Lisp コードが外部プログラムを呼び出し、 CFN/DUN 生成ツールと PDN 生成ツールが対象プログラムの解析を行って、解析結果を Emacs に返すとともに、ファイルにも出力する。

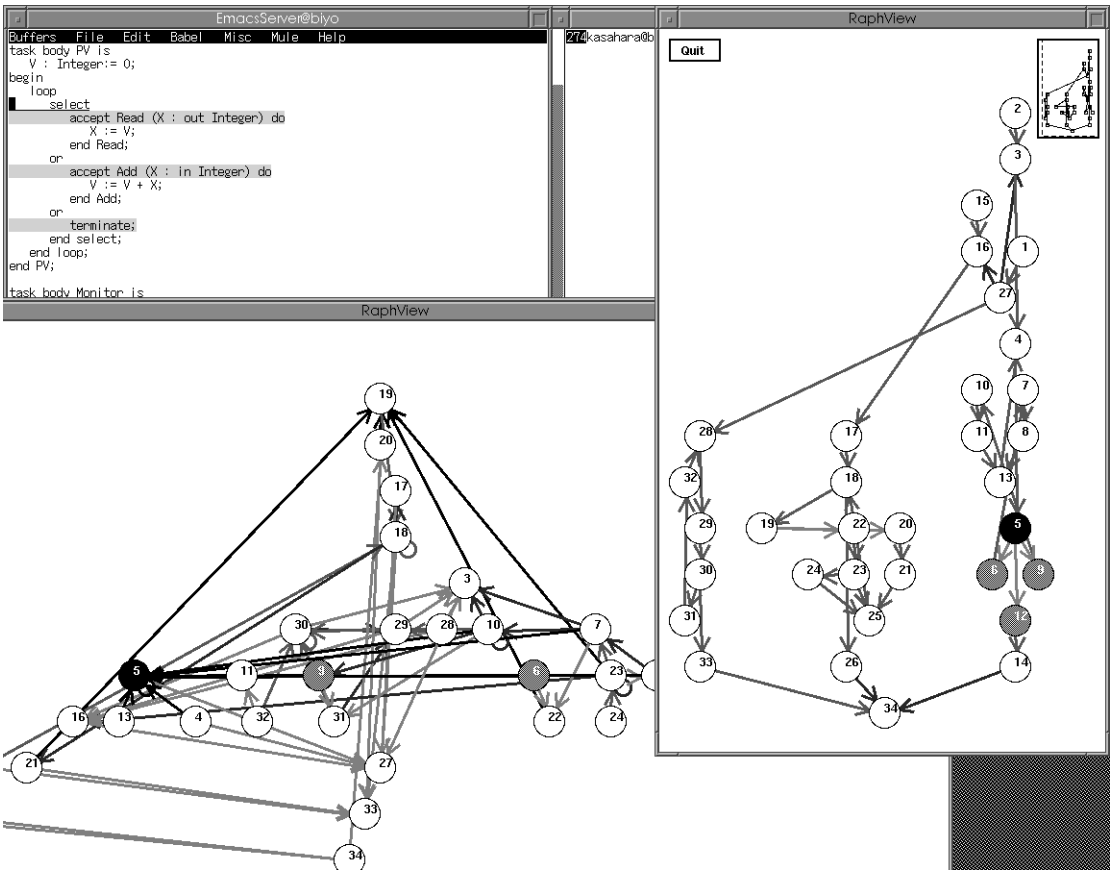

図 4.7: グラフ / ネット表示ツールの起動

図 4.7は、 CFN/DUN 生成ツールと PDN 生成ツールによって得られた PDN と CFN をグラフ / ネット表示ツールでグラフ的に表示した画面である。右側が CFN、下が PDN である。この図の中で、ソースリスト編集 / 表示ツール側は制御の流れを追跡する機能を使って、カーソルの合っている select 文の次に実行される可能性のある文を強調表示している。この機能は、例えばネストした複雑な条件分岐文の分岐先をすばやく調べたりするのに利用できる。条件分岐文のネスト関係の間違いは、起こりやすい割に影響の大きいバグを引き起こすことが多く、実際に制御の流れを画面で追跡できることはこのようなバグの発見や予防に有効である。プログラムが長大で、強調されている文が画面に収まらない場合でも、カーソルを前後の強調表示している文に移動するコマンドがあるため、容易に全体を見渡すことができる。逆に、次に特定の文を実行する可能性のある文を強調表示することもできる。これも、制御構造の間違いを探すのに役立つ。これらの処理には CFN の情報を使っている。ソースリスト編集 / 表示ツールと同期して、グラフ / ネット表示ツールの対応するノードも着色される。また、

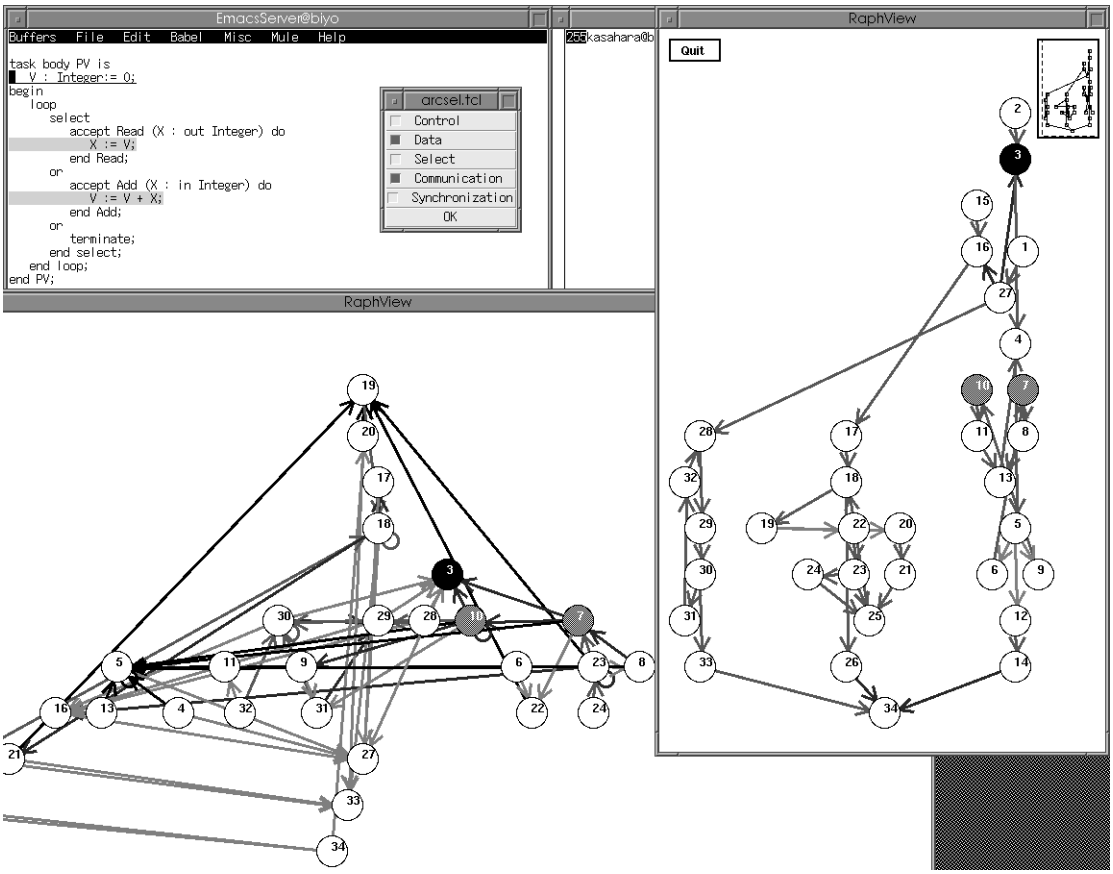

図 4.8: 直接従属している文の表示

グラフ / ネット表示ツールの好きなノードをマウスでクリックすることにより、ソースリスト編集 / 表示ツール側でそのノードに対応する文にカーソルを移動することができる。

図 4.8では、ある変数の定義がどこまで到達しているかを見ている。これは、 PDN 生成ツールで得られた PDN の情報を用いて、その変数の定義のある文に直接データ従属している文を追跡表示したものである。例えば、初期化されていない変数の宣言文でこれを行えば、未定義のまま変数を使っている場所が一目でわかり、このようなミスによるバグを未然に防ぐことができる。また、あるプロセスのある変数に関して直接通信従属している文を調べることにより、その変数が別のプロセスのどの変数と対応しているかが一目でわかる。逆に、特定の種類の従属性に関してある文が直接支配している文を参照することもできる。画面の上の方に出ている小さなウィンドウは、 5 種類の従属性のうち従属性の追跡に使いたい種類を選ぶためのものである。現在この選択はユーザにまかされている。実際にはある特定の従属性の組み合せには何らかのプログラム上の意味があると考えられるため、今後、組み合せの持つ意味に関する

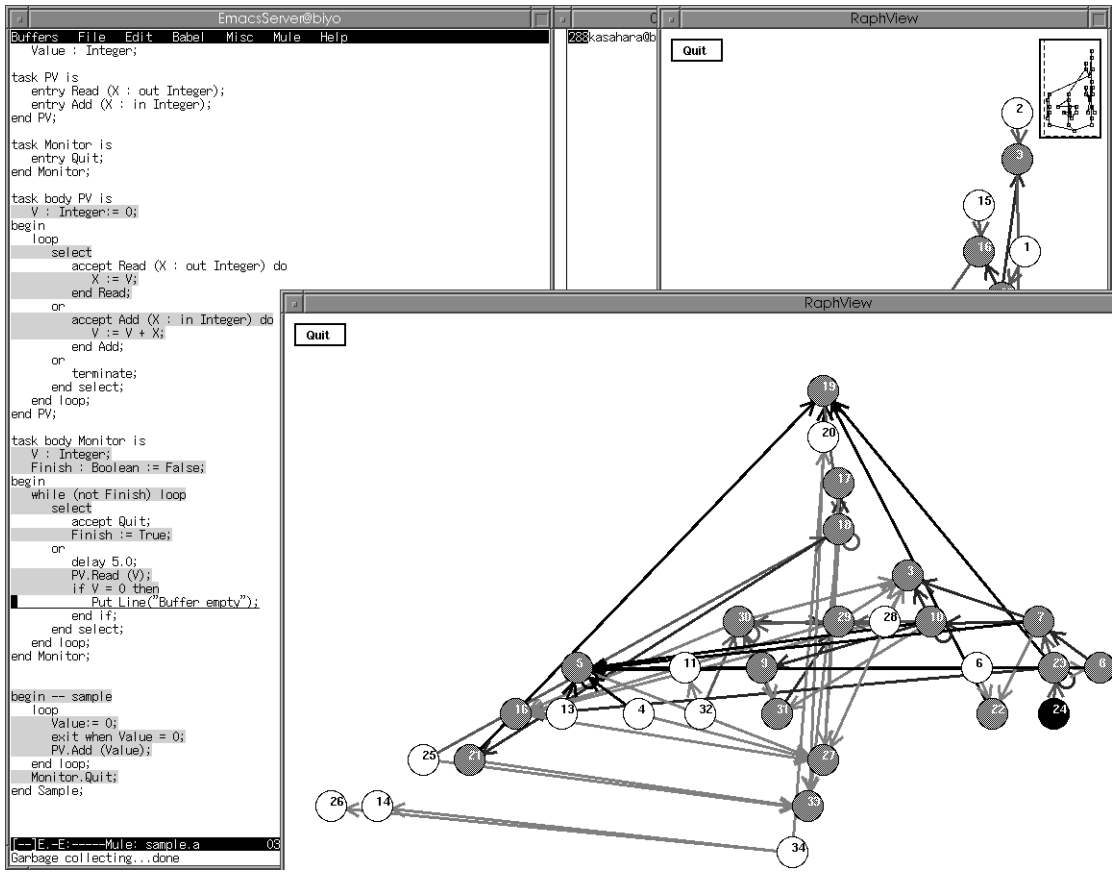

図 4.9: 後方スライスの追跡

研究を進めることにより、意味のはっきりした物に関してはメニューに組み入れる予定である。また、ユーザがよく利用する組み合せをメニュー等に登録したりできるようにすることも考えている。

図 4.9では、プログラムスライス生成ツールが PDN にもとづいて計算した、カーソルのある文からの全従属性に関する後方スライスを表示している。これにより、ある文の実行に影響を与える可能性のある文を調べることが容易になる。これは、デバッグの際にバグの存在する可能性のある文を絞り込むのに利用でき、デバッグにかかる時間を短縮できる。逆に、前方スライスを利用できれば、その文の実行が影響を与える文を見ることができる。これは保守の際に有用な情報であり、プログラムの変更によって変化した従属関係から新たなバグの発生を防いだり早期に発見したりするのに役立つ。スライスを生成する際にも、考慮したい従属性の種類を指定することができる。これにより、データの流れだけに注目したスライスや、制御の相関関係だけに注目したスライスの利用といったことができる。現在は静的な前方・後方スライス

のみをサポートしているが、今後実行モニタと組み合わせることにより動的スライスにも対応する予定である。また、従属性の追跡やスライスの追跡によって強調表示されている文だけを別ファイルに記録することができる。これは別の開発支援ツールで利用することになる。

この例では全て Ada プログラムを使っているが、C および Pascal プログラムでもまったく同じ操作系で同じ処理を行うことができるようになっている。ソースリスト編集 / 表示ツール、プログラムスライス生成ツール、グラフ / ネット表示ツール側には言語依存な処理は一切ない。このことから、CFN、DUN、および PDN による言語に依存しない統一的プログラム表現の効果が確認できる。

4.4　まとめと今後の課題

本研究では、統一的プログラム表現にもとづく逐次・並行プログラムの統合的開発支援環境を提案し、これにもとづいていくつかの支援ツールの開発と統合を行った。

今後の課題としては、次に述べるような問題がある。

まず、統一的プログラム表現にもとづいて、さらに多くのソフトウェア開発・保守支援手法の開発、ツールの実装と環境への統合を行う必要がある。現在はまだ統合的開発支援環境全体の構想のうちの一部のツールを試作した段階である。今後、その他の支援ツールについても設計、開発を行う必要がある。また、現在の統一的プログラム表現には表現力に不十分な点があるため、開発支援に必要な要求を反映して表現自身の改良が必要である。比較的開発方針が見えているツールとしては、複雑さ計測ツール、スライスを用いた比較的単純なデバッグ支援ツール、および各言語ごとの実行モニタがある。保守支援ツール、テスト支援ツール、知的なデバッグ支援ツール等の開発にはさらなる基礎研究が必要である。また、現在提案しているツールはほとんど開発活動の下流にあたる部分を占めており、いわゆる上流 CASE にあたる部分が欠けている。この部分を埋めるためには、プログラム従属性にもとづく要求定義分析法や、形式的仕様に対する従属性を用いた検証法等の研究が必要である。

また、本環境の有効性の検証と性能評価を行う必要がある。 現在、統合的開発支援環境の一部のツールの開発を行っているが、この環境が実際に実用規模のプログラム開発にどの程度有効であるかを調べる必要がある。また、対象プログラムを解析して統一的プログラム表現に変換する際の速度、メモリ消費量等について定量的な評価を行い、改良等を行う必要がある。

第 5 章

Ada 並行プログラムにおけるデッドロックの自動検出

この章では、統合的開発支援環境を構成するツールの一つである並行プログラムのデッドロック検出ツールについて述べる。デッドロックは並行プログラムにとって重大な問題であり、並行プログラムの開発支援においてデッドロックの検出は必須であると考えたため、特に重点的に研究を行った。デッドロック検出ツールは言語依存であるが、特にこの論文では Ada 並行プログラムのデッドロックを対象としたツールについて述べる。

プログラミング言語 Ada には、 1983 年に ANSI Ada 標準として認定され、 1987 年に ISO 標準となった Ada83 と、 Ada83 の改訂版で 1995 年に ISO 標準になった Ada95 がある。この章では、まず Ada83 のデッドロック検出について述べる。続いて Ada83 用デッドロック検出ツールを Ada95 に対応させる際に問題となる点について考察する。

5.1　Ada83 並行プログラムのデッドロック検出

この節では、 Ada83 並行プログラムのデッドロック検出手法とこれを実装したデッドロック検出ツールについて述べる。

Ada83(以下 Ada と表記) は、 1970 年代のソフトウェア工学とプログラミング言語に関する研究成果の集大成であり、タスキングと呼ばれる強力な並行プログラミング機能を備え、現在、大規模、実時間ソフトウェアの開発で広く使用されているプログラミング言語である。

デッドロックは、並行処理システムにおける重大で、かつ複雑な問題の一つである。タスキングデッドロックとは、 Ada 並行プログラムにおいて、複数のタスクが通信または同期のために循環的な待機状態に陥り、絶対に動作できなくなった状態をいう。

タスク間の通信や同期のために、 Ada のタスクには 5 種類の待機状態が存在する。起動待機、エントリ受付待機、エントリ呼出待機、依存性待機、終了待機の 5 種類である [40]。この各待機状態のさまざまな組み合わせによって、さまざまな種類のタスキングデッドロックが発生しうる。 5 種類の待機状態の組合せの数は 31 種類になるが、そのうち実際にデッドロックになり得るのは 18 種類であることが分かっている [7]。

並行処理システムにおいて、デッドロックは避けては通れない問題であり、発生しないように予防、予測して回避、または発生したデッドロックを検出し解消する処置が必要である。Ada 並行プログラムにおいても同様のことがいえる。そのため、タスキングデッドロックの検出法および検出ツールについて多くの論文が発表された [9]。これらの方法は動的検出法と静的検出法の 2 種類に大きく分類することができる。

理想的なタスキングデッドロックの検出法は次の二つの条件を満足するべきであろう。
(1) 任意の Ada プログラムの中に存在する任意のタスキングデッドロックを有限時間内に発見できなければならない。
(2) 任意の Ada プログラムにおいて、その中に存在しないタスキングデッドロックを報告してはいけない。
しかし、今まで発表されたどの方法も、任意の Ada プログラム中に発生する 18 種類のタスキングデッドロックを全て扱うことはできないため、上記の条件を満たしているとはいえない。任意の Ada プログラム中に発生する全てのタスキングデッドロックを検出する方法は、今だに未解決の問題である [9]。

われわれは、すでに、タスキング状態の形式モデルである「タスク待ちグラフ」を構築し、それにもとづいてタスキングデッドロックを動的に検出する方法を検討した [8]。本研究は、その原理にもとづいて、実際にタスキングデッドロックを動的に検出するツール Tasking deadlock detector を開発することを目的とする。このツールは、対象プログラム中の各タスクの振舞いを監視し、その情報によってタスク待ちグラフを作成し管理することにより、 Ada プログラム中に発生しうる 18 種類のタスキングデッドロックを全て検出することができる。

5.1.1　タスキングデッドロック動的検出の原理

タスク間の待機関係

Ada は、タスク間に、次の 5 種類の待機関係を定義している [40]。

(1) 起動待機 (Activation waiting) : タスク本体は、対応するタスク型のタスクオブジェクトによって指されるタスクの実行を定義する。この実行の最初の部分は、タスクオブジェクトの起動と呼ばれる。起動は、タスク本体の宣言部の確立から成る。タスクオブジェクトを宣言するオブジェクト宣言が宣言部のすぐ内側にある場合、タスクオブジェクトの起動は、その宣言部の確立のあとに始まる。宣言部に続く最初の文は、これらタスクオブジェクトの起動が終ったあとで実行される。タスクオブジェクトが、割当子の評価によって作られたオブジェクトまたはそのようなオブジェクトの下位要素である場合、そのタスクオブジェクトはこの割当子の評価によって起動される。これらの起動が終った後で、割当子はこのようなオブジェクトを指すアクセス値を返す。従って、プログラム構文要素の宣言部のすぐ内側で宣言された、またはプログラム構文要素の中で評価された割当子によって生成されたタスクが起動されると、その構文要素はタスクの起動が終るまで待たされることになる。

(2) エントリ受付待機 (Acceptance waiting) : タスクが、そのエントリの呼出以前に accept 文に達する場合には、そのタスクの実行は、このような呼出が受けとられるまで待機状態になる。同様に、開いた選択肢でのランデブがすぐには可能でなく else 部もない場合、タスクは開いている選択待機選択肢を選択できるまで待機する。

(3) エントリ呼出待機 (Entry call waiting) : 呼出タスクがエントリ呼出文を実行した時、そのエントリを持つタスクが対応する accept 文にまだ達していない場合には、呼出タスクの実行は待機状態になる。エントリが呼び出され、対応する accept 文に達すると、 accept 文の文の列がもしあれば呼び出されたタスクによって実行され、呼出タスクは待機状態のまま待たされる。同様に、あるタスクが条件付き、または時限エントリ呼出を行なって、ランデブが開始されると、呼出側のタスクはランデブの終了まで待機状態になる。

(4) 依存性待機 (Subprogram and/or block termination waiting) : タスクがそれに依存するタスクを持っている場合は、そのタスクの実行が完了し、それに依存する全てのタスクが終了した時に終了する。ブロック文または副プログラム本体の実行が完了しても、それに依存する

タスクが全て終了するまで制御は他へ移らない。

(5) 終了待機 (Task termination waiting) : あるタスクが、他のタスクが終了するのを待たなければならない時がある。あるタスクが終了するのは、そのタスクの実行が select 文の開いている terminate 選択肢に達し、そのタスクの依存しているマスタの実行が完了していて、かつそのマスタに依存する各タスクがすでに終了しているか、または同じように select 文の開いている terminate 選択肢で待っている時に限られる。

時限エントリ呼出、条件つきエントリ呼出において、エントリ呼出が受け付けられなかった場合、および delay 選択肢を伴う選択待機は、相手方のタスクとランデブに入らなくても待機状態を解除できるため、デッドロックの原因とはなり得ないのでここでは考えない。

このとき、上記のタスク間の待機関係には、相手方のタスクにタスキング事象が生起しない限り待機状態が変化しないという性質がある。従って、いくつかのタスクが循環的待機関係に陥った時、タスキングデッドロックが発生しているかもしれないといえる。

静的検出法と動的検出法

タスキングデッドロックの検出には、大きく分けて、静的検出法と動的検出法の 2 種類がある。静的検出法は、対象プログラムをその数学的モデルに変換し、それを解析することによって、対象プログラムの内部に発生しうるデッドロックを報告する方法である。それに対し、動的検出法は、対象プログラムを実際に動作させて、その振舞いを監視し、内部に発生したデッドロックを報告するという方法である。以下に、それぞれの特徴を列挙する。

静的検出法

- 利点
  - その方法で発見できうる範囲において、デッドロックが検出されなかった場合、その範囲で Deadlock-free の保証が得られる。
- 欠点
  - 動的に生成されるタスクに対応が出来ない。
  - 実際には発生しないようになっているデッドロックを報告する可能性がある。

- タスクの数が増えると、極端に計算の手間が増える。

動的検出法

- 利点
  - 動的に生成されるタスクにも対応が可能。
  - 嘘をつかない、即ち、存在しないデッドロックを報告しない。
- 欠点
  - 監視動作によって対象プログラムの振舞いが変化することがある。
  - モニタを通している時デッドロックが発生しなかったからといって、モニタを外した時デッドロックが起こらないとは限らない。

本研究では、対象プログラム中に発生するタスキングデッドロックを全て検出できるために、動的検出法を用いることにした。

タスク待ちグラフ

タスキングデッドロックを形式的に扱うために、タスク間の待機状態を形式的に表現できるモデルとして、タスク待ちグラフを導入した [8][10]。

タスク待ちグラフはタスクの待機状態を表す有向グラフである。このグラフは、タスクと、副プログラムまたはブロックを表す 2 種類の節点と、先に述べた 5 種類の待機関係を表す 5 種類の枝を持つ。2 種類の節点はそれぞれ “task node”、“block node” と呼び、5 種類の枝はそれぞれ “activation-waiting arc”、“acceptance-waiting arc”、“entry-calling arc”、“dependence arc”、“termination-waiting arc” と呼ぶ。詳しい定義などは [8][10] を参照されたい。

図 5.1に、簡単な Ada 並列プログラムと、そのプログラム内で 5 つ全ての待機関係を含む Complex-tasking-blocking と呼ばれるデッドロックが発生した時のタスクの待機状態を表すタスク待ちグラフを示す。

タスク間の待機関係の性質から、タスキングデッドロックが発生した時には、必ずタスク待ちグラフに有向閉路が生じている。これは必要条件であり、タスク待ちグラフに有

```
procedure CTB is
 task T1 is entry E1; end T1;
 task T2 is entry E2; end T2;
 task T3 is entry E3; end T3;
 function GET return INTEGER is
 begin
  T2.E2;
  return 0;
 end GET;
 function LIVELOCK return INTEGER is
 begin
  loop null; end loop;
  return 0;
 end LIVELOCK;
 task body T1 is
  task T4;
  task T5 is entry E5; end T5;
  task body T4 is
  begin
   B:
   declare
    task T6;
    task body T6 is
    begin
     T5.E5;
    end T6;
   begin null; end B;
   T1.E1;
  end T4;
  task body T5 is
   task T7;
   task T8;
   task body T7 is
    I:INTEGER := GET;
   begin null; end T7;
   task body T8 is
    I:INTEGER := LIVELOCK;
   begin null; end T8;
  begin
   accept E1;
  end T1;
  task body T2 is
  begin
   select
    when FALSE => accept E2;
   or
    terminate;
   end select;
  end T2;
  task body T3 is
  begin
   select
    T1.E1;
   else
    T3.E3;
   end select;
   accept E3;
  end T3;
begin null; end CTB;
```

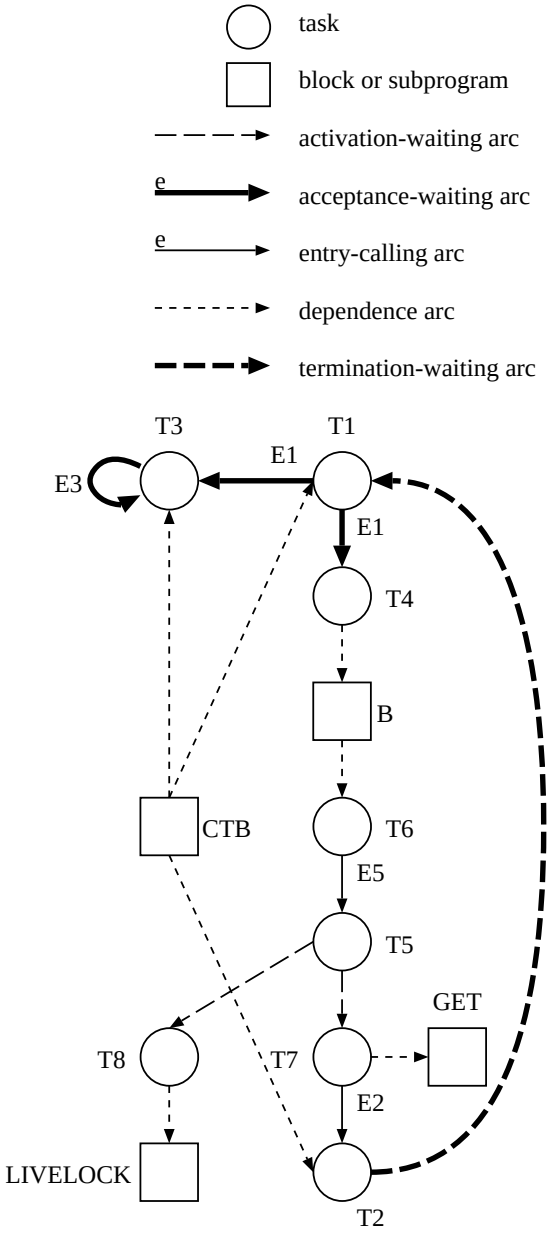

図 5.1: タスク待ちグラフの例

向閉路が生じていても、それがタスキングデッドロックを形成しているとは限らない。acceptance-waiting arc を含む有向閉路の場合、エントリ受付待機状態にあるタスクに、有向閉路に含まれないタスクからエントリ呼出がかかると、その有向閉路は解消されてしまう。従って、 acceptance-waiting arc を含む有向閉路を発見した場合、エントリ受付待機状態にあるタスクのエントリがその他のタスクから呼ばれる可能性があるかどうかを知る必要がある。

acceptance-waiting arc を含まない有向閉路は、エントリ受付待機と関係しないタスキング

デッドロックの必要十分条件である。

タスキングデッドロック動的検出の原理

任意の Ada プログラムの実行時の振舞いを正確に監視し、その中の全てのデッドロックを発見できるような動的検出法を確立すれば、 Ada タスキングデッドロック検出の問題は完全に解決すると考えられる [9]。

もちろん、動的検出法は、実行時の状態に依存するため、モニタリング中に発生しなかったデッドロックが、モニタなしで実行している時に発生する可能性もある。将来的には、対象プログラムの一部として常に動作し、デッドロックが発生した時 (または発生しそうな時) それを報告するようなツールを開発することでその問題を解決したいと考えている。

任意の Ada プログラムについて、その振舞いを正確に監視することのできる実行モニタができれば、それが提供する情報にもとづいて実行時に対象プログラムのタスク待ちグラフを生成し操作することができる。そうすれば、タスキングデッドロックの必要十分条件を満たすタスク待ちグラフ中の有向閉路を検出することにより、様々のデッドロックを検出することが可能である。タスク待ちグラフを操作し、その中の有向閉路を正確に検出するには、対象プログラムの実行中に少なくとも次のことができなければならない。

まず、モニタはそれぞれのタスクを、その生存期間中に、一意に区別できなければならない。

次に、実行中の任意のタスクについて、それが何をしているか、何をしようとしているかを知ることができなければならない。すなわち、タスク待ちグラフに影響を与えるあらゆる処理を知ることができなければならない。

3 つめに、実行中の任意のタスクについて、そのタスクがエントリ受付待機に入った時そのエントリを呼び出す可能性のあるタスクを把握することができなければならない。これは、エントリ受付待機を含むタスキングデッドロックを検出するのに必要である。

事象駆動型のタスキングデッドロック検出の原理を図 5.2に示す [8][10]。

このアルゴリズムについて簡単に説明する。

まず、このアルゴリズムにおける、タスキング事象の細かい定義や、タスキング情報の収集法などは、実現の方法に依存する。

```
task body TASKING_DEADLOCK_DETECTOR is
  Declare TWFG for target programs as a data structure and initialize it to the empty graph;
begin
  loop
  select
    accept   RECEIVE_A_TASKING_EVENT (EVENT : in TASKING_EVENT_TYPE) do
    case     EVENT is
    when     "task T is about to be elaborated by its master M_a" =>
             Add a new node T into TWFG;
             Create an empty queue for every entry of T; Set the calling task set of T to the empty set;
             if In TWFG there exists a task T_a waiting for accepting a call to entry e from T then
               Add a new acceptance-waiting arc (T_a, T, e) into TWFG;
             end if;
    when     "block or subprogram BS is about to be executed by tasking object TBS" =>
             Add a new node BS and a new dependence arc (TBS, BS) into TWFG;
    when     "task T is about to be activated by its master M_a" =>
             Add a new activation-waiting arc (M_a, T) into TWFG;
    when     "task T has been activated by its master M_a" =>
             Remove the activation-waiting arc (M_a, T) from TWFG;
             if M_a is not a task then Add a new dependence arc (M_a, T) into TWFG; end if;
    when     "task T is complete" =>
             if In TWFG there exists an acceptance-waiting arc (T_a, T, e) and
                 (T_a is involved in a directed cycle or
                 T_a is in an OR-acceptance path of an OR-acceptance node involved in a directed cycle) and then
                 The directed cycle satisfying the sufficient condition for a deadlock then
               Report the deadlock;
             end if;
             Remove the node T from TWFG;
    when     "execution of block or subprogram BS terminates" =>
             Remove the node BS from TWFG;
    when     "task T is about to reach an accept or a selective wait with an open accept alternative for entry e" =>
             if The queue of entry e is empty then
               Add a new acceptance-waiting arc (T, T_c, e) into TWFG
               for every task T_c in e's communication-dependent task set if T_c has been elaborated;
             end if;
             if In TWFG there exists a directed cycle satisfying the sufficient condition for a deadlock then
               Report the deadlock;
             end if;
    when     "task T_c is about to call entry e of task T by a simple entry call" =>
             Add T_c into the queue of entry e of T and the calling task set of T;
             Add a new entry-calling arc (T_c, T, e) into TWFG;
               if In TWFG there exists a directed cycle satisfying the sufficient condition for a deadlock then
               Report the deadlock;
             end if;
    when     "task T_c is about to call entry e of task T by a conditional or timed entry call" =>
             Add T_c into the queue of entry e of T;
    when     "a call to entry e of task T issued by task T_c has been canceled" =>
             Remove T_c from the queue of entry e of T;
    when     "rendezvous between tasks T_c and T for entry e of T has started" =>
             Record the state of T_c as "Suspended-by-rendezvous";
             Remove all acceptance-waiting arcs of the form (T, T_s, e) from TWFG;
             if In TWFG there exists no entry-calling arc (T_c, T, e) then
               Add a new entry-calling arc (T_c, T, e) into TWFG;
             else
               Remove T_c from the calling task set of T;
             end if;
    when     "tasks T_c and T complete their rendezvous for entry e of T" =>
             Remove T_c from the queue of entry e of T; Remove the entry-calling arc (T_c, T, e) from TWFG;
             Record the state of T_c as "Working";
    when     "task T is about to reach a selective wait with a terminate alternative as the only open alternative" =>
             Add a new termination-waiting arc (T, T_s) into TWFG
               for every task T_s which depends on the same master as T;
             if In TWFG there exists a directed cycle satisfying the sufficient condition for a deadlock then
               Report the deadlock;
             end if;
    when     "task T_a is about to be aborted" =>
             Remove all entry-calling arcs of the form (T, T_a, e) from TWFG;
             Remove T_a from the queue of every entry of every task and the calling task set of every task;
             if In TWFG there exists an entry-calling arc (T_a, T, e) and
                 state of T_a is not "Suspended-by-rendezvous" then
               Remove the entry-calling arc (T_a, T, e) from TWFG;
             end if;
             if In TWFG there exists a directed cycle satisfying the sufficient condition for a deadlock then
               Report the deadlock;
             end if;
    when     others => null;
    end case;
    end RECEIVE_A_TASKING_EVENT;
  or
    terminate;
  end select;
  end loop;
end TASKING_DEADLOCK_DETECTOR;
```

図 5.2: 事象駆動型のタスキングデッドロック検出アルゴリズム

つぎに、このアルゴリズムでは、エントリ呼出に関する情報が必要である。あるタスクがエントリ受付待機状態に入った時、そのエントリを呼び出す可能性のあるタスクとの間に、

Acceptance-waiting arc を付け加えなければならないからである。ある時点でどのタスクがどのエントリに対しエントリ呼出をかけることができるかということを知るのは非常に困難である。そこで、全てのタスクの中で、問題のエントリへのエントリ呼出文を持っているタスク全てに Acceptance-waiting arc をつなぐことにする。この情報は、対象プログラムを静的に解析することによって得られる。

タスキングデッドロックを検出するためには、デッドロックの必要十分条件を満たす有向閉路がタスク待ちグラフ中に存在するかどうかを知ることができなければならない。タスク待ちグラフ中の有向閉路を発見するのは容易である。問題は、その有向閉路に中に acceptance-waiting node がある時である。この時は、 Hermann と Chandy の AND-OR デッドロックモデルにおける木操作アルゴリズムによってデッドロックかどうかの決定が可能である [27]。

このアルゴリズムは、エントリ受付待機を含まないデッドロックに関しては、その発生の直前に報告することができる。エントリ受付待機を含む場合、そのデッドロックを形成している有向閉路の外からエントリ呼出がかかるかどうかを、デッドロックが発生した瞬間に決定することは不可能であるから、問題のエントリに対するエントリ呼出文を持つタスクが全て動作しなくなるまで報告することはできない。

今のところ、 abort 文が実行された時の処理はなされているが、 abort 文が実行される可能性については考慮していない。従って、存在しないデッドロックを報告することはないが、報告したデッドロックがいつか解消される可能性はある。

5.1.2　ツールの実現

モニタリングの原理とツールの構成

Ada 並行プログラムの実行時の振舞いを監視する方法として、 2 つの方法が考えられる。一つは、対象プログラムをソースリストレベルで変換する方法である。この方法では、プリプロセッサによって、監視の対象となるプログラム P をプログラム P' に変換する。 P' は P とタスクの振舞いにおいて等価であり、実行中に P で発生するタスキング事象を通信によって実行時モニタに報告するものである。もう一つの方法は、実行時のランタイム情報を利用するものである。この方法では、実行時モニタは Ada 処理系の実行時の環境の一部として実現され、実行

時モニタは対象プログラムのタスキング事象を直接受けとることができる。本研究では、高い移植性を実現するために、ソースリストを変換する方法を使って Tasking deadlock detector を実現することにした。

並行プログラムを監視する場合、対象プログラムの振舞いは実行時の環境に左右されるため、いわゆる「不確定性原理」が働く。すなわち、監視機構によって、対象プログラムの振舞いが影響を受け、もとのプログラムと違ってくるかも知れないのである。従って、実行時モニタを実現する際には、いかにして対象プログラムへの影響を少なくするかが重要な要件になる。われわれは、並列プログラムの監視に対して、「半順序関係保存」という概念を導入した [5][11]。これは、実行時モニタが対象プログラムの振舞いに与える影響を少なくする時の、最小限の要求である。

われわれは、すでに “EDEN” と名付けられた Ada 並行プログラム用事象駆動型実行モニタを開発している [6]。 EDEN は半順序関係保存を念頭においた実行時モニタであり、対象プログラム内のいくつかのデッドロックを検出することができる。今回開発したツールは、 18 種類全てのデッドロックを発見できるもので、 EDEN 改訂版の一部として実現した。図 5.3に EDEN を使用した Ada プログラム監視の原理を示す。

EDEN は大きく分けてプリプロセッサと実行時モニタの 2 つの部分からなっている。対象プログラムは、プリプロセッサによって変換され、同時にエントリ呼出表が生成される。変換された対象プログラムは Ada コンパイラによってコンパイルされ、別にコンパイル済みの実行時モニタおよびその他のライブラリとリンクされ、実行形式にされる。これを実行することにより、内部で、変換された対象プログラムが起動され、対象プログラムは実行時モニタにタスクの振舞いに関する情報を送る。 Tasking deadlock detector は、プリプロセッサによって生成されたエントリ呼出表と実行時モニタが収集したタスキング情報をもとにタスク待ちグラフを生成し、操作する。そして、タスキングデッドロックを検出すると、端末を通してユーザにタスキングデッドロックに陥る (または陥っている) タスクおよびその状態を報告する。 Tasking deadlock detector の構成を図 5.4に示す。

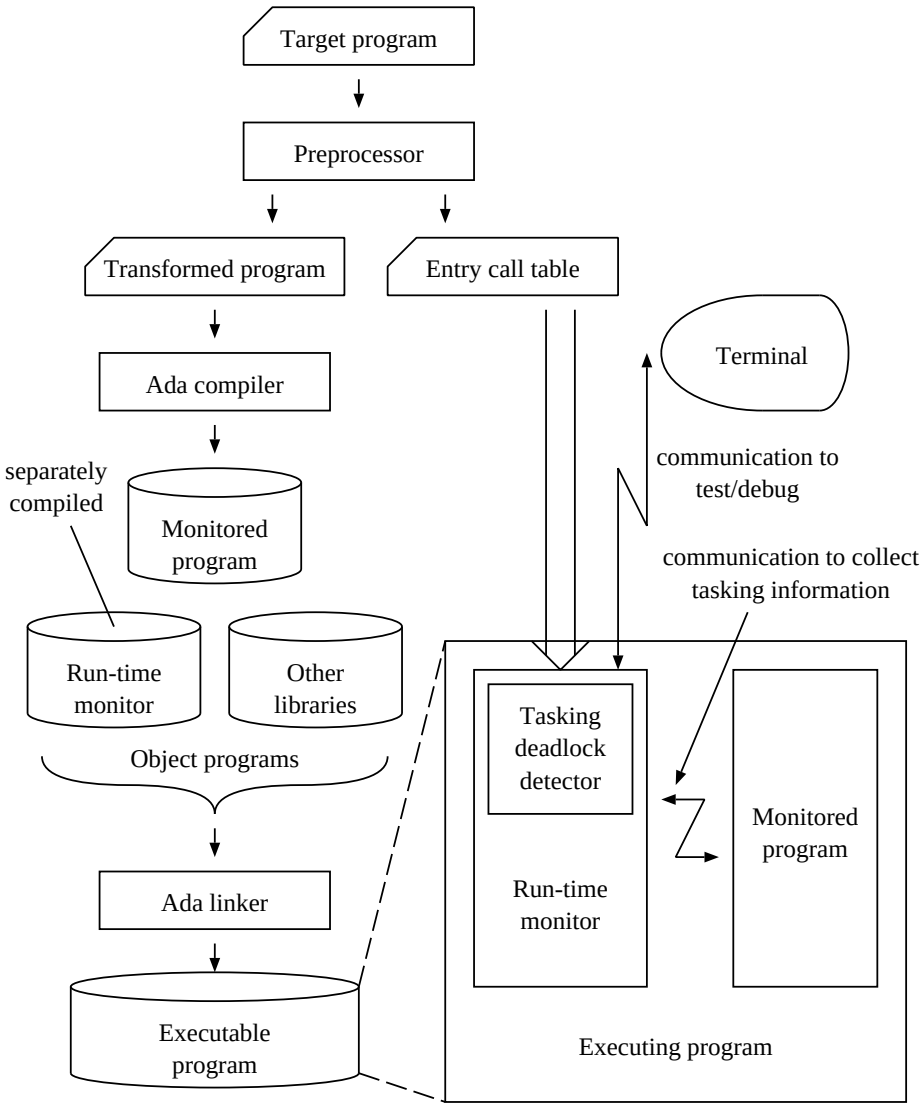

図 5.3: EDEN を用いたモニタリング過程

タスクの一意的な命名法

タスクの振舞いを監視し、正確にタスク待ちグラフを操作するためには、それぞれのタスクを区別できるように、各タスクに一意的な名前づけを行なわなければならない。ところが、Ada には、タスクが自分自身の名前を知ることができるような命令がないため、タスクに一意的な名前づけを行なうためには、対象プログラムを適当に変換し、名前を与えてやるようにしなければならない [4]。

本研究では、まず German の方法について考察した [21]。German のタスク命名法の基本的な考え方は、各タスクに、例えば SET_ID というエントリを追加し、そのタスクの親がエント

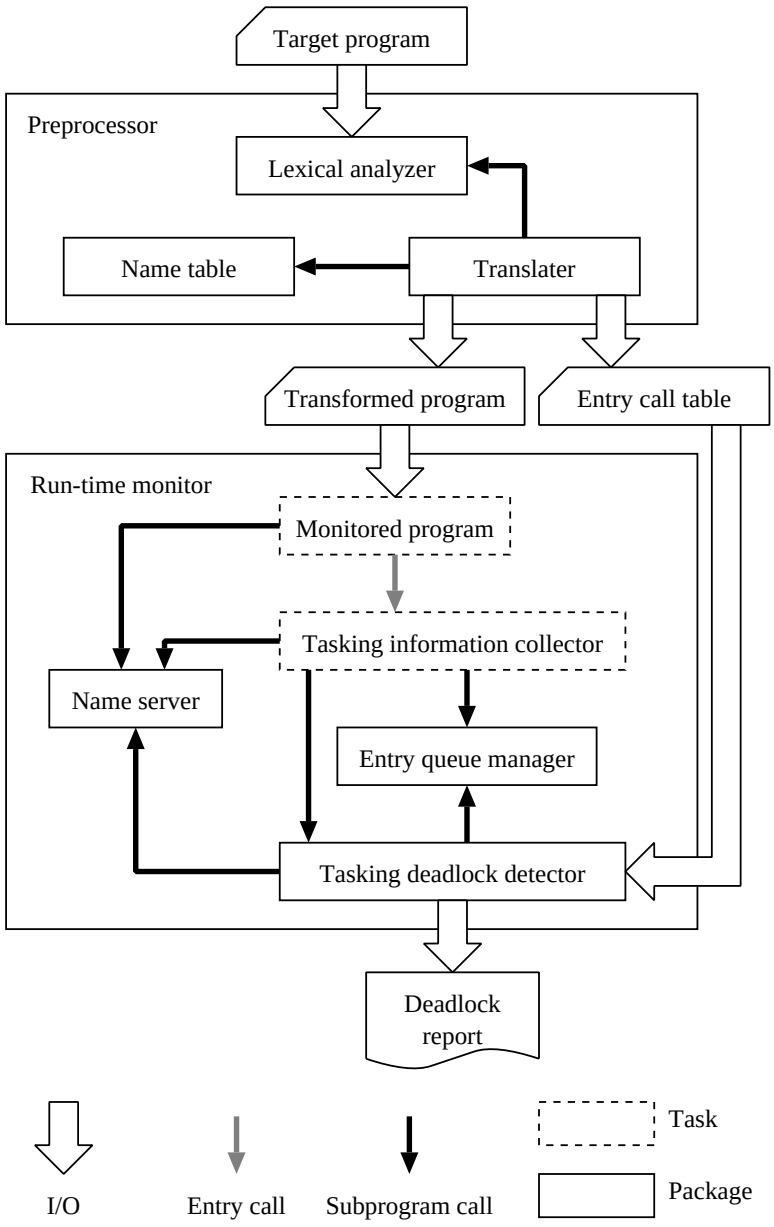

図 5.4: Tasking deadlock detector の構成

リ呼出によってそのタスクの内部名を渡してやる、というものである。タスクの起動時にもその内部名が参照できるようにするために、タスク本体の宣言部を内側のブロックに入れるという方法をとっている。ところが、この方法には、タスキングデッドロック検出に使うには致命的な欠陥があることが分かった。このような変形を行なうと、タスクの振舞いがもとのプログラムと違ってくるのである。タスク本体の宣言部が一つ内側のブロックに移ってしまうために、タスクがそのブロック中で宣言された別のタスクの起動を待たなくなってしまい、結果的

に、もとのプログラムでは発生し得る、起動待機関係を含む全てのデッドロックは発生しなくなってしまう。

現在、本ツールでは、旧 EDEN で使用されたタスクの命名法を若干変更したものを使用している。この方法は、各タスクが、その起動時に、大域的なタスクとして実現された名前サーバのエントリを呼び出して、自分の内部名を受けとるというものである [4]。

現在使用している方法の問題点は、 1 つのタスク型が、複数のタスクを生成する場合には適用できないという点である。このため、今回開発したツールでは、全てのタスクが単一タスクである Ada プログラムしか扱うことができない。この命名法を改良して、任意の Ada プログラムに適用できるようにすることは、今後の研究課題である。

無論、この問題はソースリストレベルでの対象プログラムの変換を用いてタスクに一意的な名前づけをする際に起こるのであって、実行時のランタイム情報を使ってタスクを識別する場合には問題はないと考えている。

タスクの情報収集

対象プログラムの状態変化に合わせてタスク待ちグラフを更新するためには、全てのタスクについて、各タスクが何をしようとしているか、また何をしているかを知ることができなければならない。

EDEN の情報収集部は、それぞれのタスキング事象が異なったエントリに対応したタスクとして実現されている。対象プログラム中のタスキング事象が発生する位置に情報収集部へのエントリ呼出を挿入するように、プログラム変換規則は作られている。様々の情報は、エントリ呼出の引数として情報収集部に渡される。

タスク待ちグラフの更新は、情報収集部においてタスク待ちグラフの更新が必要なタスキング事象発生の報告を受けるエントリの中で、 Tasking deadlock detector パッケージ中のタスク待ちグラフに変更を加える副プログラムを呼び出すことで行なわれる。これがタスクでなく、副プログラムの形をとっているのは、対象プログラム中のタスクの状態とタスク待ちグラフの状態が常に対応するようにするためである。副プログラムにしておけば、タスク待ちグラフの更新中は、対象プログラムの情報収集部へのエントリ呼出は待機させられるので、タスク待ちグラフの更新中に対象プログラムの状態が変化することはない。ただし、欠点として、対

象プログラムに与える影響が大きくなることが挙げられる。

エントリ呼出表

エントリ受付待機を含むタスキングデッドロックを正しく検出するためには、エントリ受付待機状態にあるエントリを呼び出す可能性のあるタスクを把握しておく必要がある。実際には、呼び出す可能性があるかどうかを予測するのは大変困難であるから、全てのタスクに対し、どのタスクがどのエントリへのエントリ呼出文を持っているかを調べておくことになる。これをエントリ呼出表と呼ぶ。

この表によって、Tasking deadlock detector は、どのタスクがどのタスクのエントリを呼び出す可能性があるかという情報を得ることができる。

エントリ呼出表は、プリプロセッサによる静的解析によって作成されるため、呼出側のタスクはタスク型で記録される。また、タスクが副プログラムを呼び出していて、その副プログラムの中で別のタスクのエントリを呼び出している場合に対応するために、タスクの副プログラム呼出、および副プログラムのエントリ、または別の副プログラムの呼出も記録している。図 5.5にエントリ呼出表の例を示す。これは、図 5.1の Ada プログラムに対するエントリ呼出表である。

エントリ及び副プログラム

タスク型及び副プログラム	T2.E2	T5.E5	T1.E1	GET	LIVELOCK	T3.E3
GET	1	0	0	0	0	0
T6	0	1	0	0	0	0
T4	0	0	1	0	0	0
T7	0	0	0	1	0	0
T8	0	0	0	0	1	0
T3	0	0	1	0	0	1

図 5.5: エントリ呼出表の例

エントリ呼出表中の各タスク、副プログラムはその識別子で記録されているため、あるタスクがエントリ受付待機にはいったとしても、まだ起動しておらず、名前のないタスクには

acceptance-waiting arc をのばすことができない。従って、新しいタスクが起動する度に、エントリ呼出表をチェックし、新しいタスクが呼び出す可能性のあるタスクのエントリを確認する必要がある。

また、タスクが完了するなどしてタスク待ちグラフ中から外される時には、そのタスクがなくなることによって呼び出される可能性が減ったエントリに対し、デッドロックのチェックをする必要がある。

Hermann と Chandy の AND-OR デッドロックモデルのための木操作アルゴリズム [27] はまだ実現するに至っていない。これは今後の課題である。

対象プログラムの制限

本ツールは、 18 種類全てのタスキングデッドロックを検出できるが、対象プログラムに 2 つの制限がある。 1 つは、対象プログラムが livelock-free でなければならない点である。もう 1 つは、対象プログラムが、 2 つ以上のインスタンスを持つタスク型を含んではならないという点である。

5.1.3　タスキングデッドロックの検出例

今まで提案されてきた方法では検出できなかったタスキングデッドロックを、本ツールにより検出した例を示す。対象プログラムは 5.1.1節の図 5.1に示した。このプログラムは 2 つのタスキングデッドロックを含んでいる。 1 つは、タスク T3 が自分自身のエントリ E3 を呼び出しているもっとも簡単なデッドロックである。もう 1 つは、 5 種類全ての待機状態を含む complex tasking deadlock と呼ばれるタスキングデッドロックである。

図 5.6が Tasking deadlock detector による complex tasking deadlock の検出結果である。これによって、タスキングデッドロックが発生した時、どのタスクがデッドロックに含まれているのか、各エントリのキューはどうなっているのかを知ることができる。

```
***   EDEN Ver. 2.1   26-Apr-1991      date:27-Apr-1991   time: 2:30:39   ***

    * A tasking deadlock will occur in the program.

    * At    54.58000000 sec after execution start,
      there is such a situation as follows :

    * task T2 is waiting for terminating together with task T1
    * task T1 is waiting for accepting a call to its entry E1 from task T4
    * task T4 is executing block B
    * block B created task T6 and is waiting for termination of task T6
    * task T6 is calling entry E5 of task T5
    * task T5 is waiting for activation completion of task T7
    * task T7 is calling entry E2 of task T2

***   EDEN Ver. 2.1   26-Apr-1991      date:27-Apr-1991   time: 2:30:39   ***

    * At    54.58000000 sec after execution start,
      states of all tasks are as follows :

    * task name             => MAIN_TASK
    * task ID of the task   =>  1
    * state of the task     => BLOCK_COMPLETED
    * calling subprogram    => CTB

    * task name             => T3
    * task ID of the task   =>  2
    * parent of the task    => CTB
    * state of the task     => SIMPLE_ENTRY_CALLING
    * communication entry   => T3 . E3

    * task name             => T2
    * task ID of the task   =>  3
    * parent of the task    => CTB
    * state of the task     => TERMINATION_WAITING

    * task name             => T1
    * task ID of the task   =>  4
    * parent of the task    => CTB
    * state of the task     => ACCEPTING
    * communication entry   => T1 . E1

    * task name             => T5
    * task ID of the task   =>  5
    * parent of the task    => T1
    * state of the task     => EXECUTION_WAITING

    * task name             => T4
    * task ID of the task   =>  6
    * parent of the task    => T1
    * state of the task     => BLOCK_COMPLETED
    * executing block       => B

    * task name             => T8
    * task ID of the task   =>  7
    * parent of the task    => T5
    * state of the task     => WORKING_FOR_INTERNAL_AFFAIRS
    * calling subprogram    => LIVELOCK

    * task name             => T7
    * task ID of the task   =>  8
    * parent of the task    => T5
    * state of the task     => SIMPLE_ENTRY_CALLING
    * communication entry   => T2 . E2

    * task name             => T6
    * task ID of the task   =>  9
    * parent of the task    => B
    * state of the task     => SIMPLE_ENTRY_CALLING
    * communication entry   => T5 . E5

***   EDEN Ver. 2.1   26-Apr-1991      date:27-Apr-1991   time: 2:30:40   ***

    * At    54.58000000 sec after execution start,
      states of all entry queues are as follows :

    * entry name => T3 . E3
    *          1 => T3

    * entry name => T2 . E2
    *          1 => T7

    * entry name => T1 . E1
    *            => the entry queue is empty

    * entry name => T5 . E5
    *          1 => T6
```

図 5.6: デッドロックの検出例

5.1.4　今後の課題

タスキングデッドロック動的検出の原理と、それにもとづいて動作する、 Tasking deadlock detector の実現について述べた。そして、具体例として、このツールが、今まで提案されてきた方法またはツールでは検出できなかった complex tasking deadlock を検出できることを示した。われわれはすでに、このツールが、全てのタスク型が単一タスクである Ada プログラム中に発生しうる 18 種類のタスキングデッドロックを全て検出できることを確認した [25]。

今後の課題として、まず第一に、このツールを任意の Ada プログラムに適用できるように拡張することが挙げられる。そのためには、任意の Ada プログラムについて、その中の全てのタスクに対する一意的な命名が可能な方法およびそのためのプログラム変換規則を開発する必要がある。現在のところ 1 つのタスク型が複数のタスクを生成する場合には対応できない。よって、タスク型の配列、レコード型、アクセス型などを使用した Ada 並列プログラムに対応できない。これは今後の大きな研究課題の一つである。

タスク待ちグラフ中に acceptance-waiting arc を含む有向閉路が発生した時、その有向閉路がタスキングデッドロックの必要十分条件を満たしているかを判別するための、 AND-OR デッドロックモデルにおける木操作アルゴリズムの実現も今後の研究課題の一つである。

今一つの課題は、処理速度の問題である。対象プログラムへの影響をより少なくするためには、アルゴリズムの改良などにより処理時間をできるだけ小さくする必要があると思われる。

しかし、対象プログラムの振舞いに対する影響を完全になくすことはできないと考えられる。従って、本研究で用いた方法では、実時間システムに用いられるようなプログラムのタスキングデッドロック検出に用いることは難しい。監視による影響を受けずに、いかにしてそのようなプログラム中のタスキングデッドロックを検出するかは、現在も未解決の問題である。

Ada のタスキング機構は複雑であり、そのため、タスキングデッドロックおよびその検出法も複雑なものとなっている。タスキングデッドロック検出の問題を完全に解決するためには、さらに研究を続ける必要がある。

5.2　Ada95 並行プログラムのデッドロック

この節では、前節で述べた Ada83 用デッドロック検出ツールを Ada95 に対応させる際に問題になる点について考察する。

最近、従来の Ada(Ada83) の改良版である Ada95 の言語仕様が International Standard として制定された [41]。今後 Ada によるソフトウェア開発は Ada95 の利用が主流になるだろう。

われわれはすでに Ada83 について、タスクには 5 種類の待機状態があり、これらの組み合わせによって 18 種類のデッドロックが起こることを示した [7]。また、これをもとに、 Ada83 並行プログラムの実行中に発生したデッドロックを動的に検出し報告するツールを開発した [9]。

今後 Ada95 の利用が主流になってくるならば、 Ada95 並行プログラム用のデッドロック検出ツールが必要になる。しかし Ada95 には Ada83 からさまざまな変更が加えられているため、 Ada83 並行プログラム用のデッドロック検出ツールをそのまま利用することはできない。並行実行機構にも機能の追加が行われており、これによってデッドロックの種類などが変化しているかもしれない。

本章では、 Ada83 での経験をもとに Ada95 を Ada83 と比較して、以前開発した Ada83 用デッドロック検出ツールを Ada95 並行プログラムに対応させるための変更方針について考察する。言語仕様の変化のうち、タスキング機構に関係があるものとしては、 protected object と `requeue` 文が新たに加わったこと、タスクの識別を支援する標準ライブラリが規格に規定されたことがあげられる。

5.2.1　待機状態の種類

Ada83 のタスクには、起動待機、エントリ受付待機、エントリ呼出待機、依存性待機、終了待機の 5 種類の待機状態があった。 Ada95 についてタスキング機構を調査した結果、この 5 種類の基本的な待機状態については Ada83 から変化がないという結論に達した。従って、Ada83 と同じく 18 種類のデッドロックが存在するとみなしてよい。

5.2.2　protected object

protected object とは、 protected operation を通して複数のタスクで共有できる資源を記述するものである。 Ada83 には存在しない。 protected object では、複数のタスクから資源に対し読み出しのみを並行に行う機構として protected function、読み書きを排他的に行う機構として protected procedure と protected entry が提供されている。 1 つの protected object に対する protected function を除く protected operation は同時に 1 つしか実行されない。複数のタスクから呼び出しが起きている場合には残りのタスクは今実行されている protected operation が終了するまで待たされることになる。

protected operation の実行中に待機状態に入る可能性のある操作を行おうとすることは言語仕様で bounded error と定義されており、処理系はコンパイル時または実行時にこれを検出して Program Error 例外を発生することが期待されている。すなわち、 protected operation の中で別のタスクを呼び出したり、タスクを生成したりするといった操作はできない。このため、 protected object を通してデッドロックを起こすことはできない。従って、 Ada95 に protected object が導入されたことによってデッドロックの種類は増えないことになる。

しかし、 bounded error をコンパイル時や実行時に検出することは必須ではないため、処理系によっては無視することもありうる。このためデッドロック検出ツール側でどのタスクが現在どの protected object を実行しているかを監視し、その中で待機状態に入る可能性のある操作を行おうとした場合ユーザに警告するべきだと考える。

ツールの実装としては、各タスクの各 protected operation の前後にタスクの識別子とともに protected object 名と operation の種類を実行モニタに渡す文を配置し、 protected object 側に自分の名前と自分を実行しているタスクの識別子等を実行モニタに渡す文を配置することにより、どのタスクがどの protected object を実行しているか、また実行可能になるのを待っているかを把握できる。また待機状態になる可能性のある文は言語仕様に定義されているので、これらの文が実行される直前にモニタにそれを伝えることにより、 bounded error の検出ができる。

5.2.3　`requeue` 文

`requeue` 文は、あるタスクが別のタスクのエントリを呼び出している時に、そのエントリの

accept 文内で実行される命令である。 Ada83 にはない。 requeue 文が実行されると、それを含む accept 文はそこで終了し、呼び出していたタスクは requeue 文で指定されたエントリのキューにつながれる。この文は新たな種類の待機状態を導入するものではないため、新しい種類のデッドロックが起こることはない。

ツールの実装としては、 requeue 文の直前にどのタスクがどこからどこに requeue しようとしているかをモニタに伝える文を加えることにより、 requeue が起ころうとしていることを知ることができる。

5.2.4　タスクの識別

デッドロック検出のためには、タスクの振舞いを監視し、その状態変化を記録する必要がある。われわれが開発した Ada83 用のデッドロック検出ツールでは、各タスクが監視プログラムに自分の状態を報告するという実装になっている。このためには、各タスクが自分の名前を知っていて、自分の名前と状態を報告する必要がある。しかし、 Ada83 にはタスク自身が自分の名前などの情報を知る手段がない。普通のタスクならば、ソースリスト中に直接名前を埋め込むことで解決できる。ところが、タスク型の配列の場合、同じソースリストから複数のタスクの実体を作ることができるため、この方法は使えない。このため Ada83 用のデッドロック検出ツールではタスク型の配列に対応できなかった。

これに対し、 Ada95 では、システムプログラム用標準ライブラリとしてタスクの識別とタスクの属性に関するライブラリが定義されており、これを用いることにより容易にタスクの名前などを取得することができる。このため、 Ada95 については、このライブラリを利用することによりタスク型の配列を含むプログラムにも対応できる。

ただし、このライブラリは Ada95 の処理系に必須ではないため、場合によっては利用できない可能性がある。この場合は Ada83 で用いていた手法を流用せざるをえない。このため、対象プログラムの変換ツールと実行モニタをそれぞれ 2 種類用意し、ユーザに選択させるか、インストール時に自動判別して適当な方を使うようにする必要がある。

5.2.5　今後の課題

Ada95 を Ada83 と比較して、以前開発した Ada83 用デッドロック検出ツールを Ada95 並行プログラムに対応させるための変更方針について考察した。

今回述べた以外にも、文法的・意味的な変更がかなり行われているが、タスキング機構に関係ない部分は割愛した。 Ada95 用のデッドロック検出ツールを開発する場合には、そのような変更点も考慮する必要がある。われわれのデッドロック検出ツールでは、最初にソースリストを一定の規則に従って変換し、監視用プログラムとリンクすることによって実行時の監視を行うため、ソースリストの変換規則にかなり手を加える必要がある。これは技術的な問題ではあるが、最も手間がかかる作業である。

今後、今回の考察を踏まえて、 Ada95 のためのデッドロック検出ツールの開発を行っていく予定である。

第 6 章

おわりに

本研究では、主にプログラム従属性にもとづいた統一的プログラム表現を用いた、統合的ソフトウェア開発支援環境の構築を目的とする研究を行った。この章では、本論文のまとめとして、本研究の成果および今後の課題について述べる。

第 1 章でも述べたように、ソフトウェアの開発支援を効果的に行うためには、対象プログラムから開発支援に必要な情報を取り出した抽象的プログラム表現が必要である。本研究ではプログラム従属性を利用した開発支援を中心に考え、そのためのプログラム表現モデルとして、CFN、 DUN、および PDN を利用することにした。そして、これにもとづいた開発環境を試作した。主な研究成果は以下の通りである。

- 統一的プログラム表現の基礎となる CFN、 DUN、および PDN 生成ツールの開発

  CFN、 DUN、および PDN を開発支援に利用するためには、対象プログラムからこれらのモデルを生成する手法を考案し、これを実装して生成ツールを開発する必要がある。本研究では、 C、 Pascal、および Ada で記述されたプログラムを入力とし、対象プログラムの CFN/DUN を自動生成するツールを開発した。また、言語に依存しない形で表現された CFN/DUN から PDN を生成するアルゴリズムを考案し、これを実装して CFN/DUN から PDN を自動生成するツールを開発した。 CFN/DUN から PDN を生成するツールは言語に依存していないため、未対応の言語で書かれたプログラムの従属性解析を行いたい場合には、その言語で書かれたプログラムから CFN/DUN を生成するツールを作るだけでよいという特徴がある。

- 統一的プログラム表現にもとづくソフトウェア開発支援ツールの開発とその統合

  統一的プログラム表現にもとづいた、逐次・並行プログラムの統合的開発支援環境の構想を示した。そしてこれにもとづいてスライス生成ツール、グラフ / ネット可視化ツール、ソースリスト編集 / 表示ツールを開発し、統合して統合的開発支援環境を試作した。この環境は当初の期待通り C、Pascal、Ada プログラムに対し同じ操作で利用することができた。また現在までにペトリネット編集・シミュレーションツール、デッドロック動的検出ツール、実行履歴取得ツールを開発したが、まだ環境に統合されるには至っていない。

- Ada83 並行プログラムのためのデッドロック検出ツールの構築

  統合的開発支援環境開発の一環として、Ada83 並行プログラムのデッドロック検出ツールを開発した。このツールは Ada83 並行プログラム中に発生する可能性のある 18 種類のデッドロック全てを動的に検出することができた。

統合的開発支援環境の構築をテーマとした今後の検討課題としては、以下のような点が挙げられる。

- 統一的プログラム表現の拡張 (手続き、ポインタ、配列等)

  CFN、DUN、および PDN は、現在手続き呼び出しを表現することができない。実際に実用規模のソフトウェアに対する開発支援を行うためには、手続き間従属性の分析、ポインタや配列に関する従属性の分析ができなければならない。このため、これらのモデル自体の表現力の拡張が必要である。

- 統一的プログラム表現にもとづくソフトウェア開発・保守支援手法の開発、ツールの実装と環境への統合

  現在はまだ統合的開発支援環境全体の構想のうちの一部のツールを試作した段階である。今後、その他の支援ツールについても設計、開発を行う必要がある。比較的開発方針が見えているツールとしては、複雑さ計測ツール、スライスを用いた比較的単純なデバッグ支援ツール等がある。保守支援ツール、テスト支援ツール、知的なデバッグ支援ツール等の開発にはさらなる基礎研究が必要である。

- 変換ツールの性能評価と開発環境の有効性検証

  対象プログラムを解析して統一的プログラム表現に変換する際の速度、メモリ消費量等について定量的な評価を行い、場合によっては生成アルゴリズムの改良等を行う必要がある。また、現在統合的開発支援環境の一部のツールの開発を行っているが、この環境が実際に実用規模のプログラム開発にどの程度有効であるかを調べる必要がある。

- その他の言語に対応した CFN/DUN 生成ツールの開発 (Ada95、 Occam2 等)

  われわれはすでに Ada83、 C、 Pascal のための CFN/DUN 生成ツールを開発した。今後、さらに他の言語のための CFN/DUN 生成ツールを開発する。これにともなって、CFN、 DUN、および PDN がさまざまな手続き的プログラムを表現するのに十分かどうか等の検証、およびこれらのモデルの改良を行う。

- デッドロック検出ツールの開発

  本論文で行った Ada95 並行プログラムのデッドロック検出に関する問題点の検討をもとに、 Ada95 並行プログラムのデッドロック自動検出ツールを開発し、これを統合環境に組み込む予定である。また、 Occam2 並行プログラムのデッドロック検出手法の考案とその実装を行う予定である。

今後、信頼性の高い並行プログラムを快適に開発できるような開発支援環境の構築を目指して、以上に述べたような問題点を解決するための研究を進めて行く必要がある。

謝辞

本論文の最後にあたり、日頃から熱心に御指導いただいている九州大学情報工学科の牛島和夫教授に深く感謝いたします。本論文をまとめるにあたり、査読をしていただき、また懇切な助言をいただきました九州大学大型計算機センターの島崎眞昭教授、ならびに、九州大学電気工学科の和田清教授に深く感謝いたします。

九州大学情報工学科の程京徳助教授、谷口秀夫助教授、情報処理教育センターの古川善吾助教授、医学部付属病院医療情報部の坂本憲広講師には有益な御教示をいただきました。ここに記しまして感謝の意を表します。

本環境の開発において、コーディングの一部を担当していただいた蒲池正幸君、乃村能成君、西和則君に感謝いたします。また、プログラム従属性について一緒に議論した、小田朋宏君、合田和正君、趙健軍君に感謝いたします。

最後になりましたが、日頃の研究活動において貴重な助言をいただきました、片山徹郎君をはじめとする研究室の皆様に感謝いたします。

参考文献

[1] Agrawal, H. and Horgan, J. R.: Dynamic Program Slicing, *Proc. ACM SIGPLAN'90*, pp. 246–256, 1990.

[2] Agrawal, H., Demillo, R. A. and Spafford, E. H., Debugging with Dynamic Slicing and Backtracking, *Softw. Pract. Exper.*, Vol. 23, No. 6, pp. 589–616, 1993.

[3] Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[4] Cheng, J. and Ushijima, K.: Naming Ada Tasks at Run-Time, *ACM Ada Letters*, Vol. 9, No. 2, pp. 52–61, 1989.

[5] Cheng, J. and Ushijima, K.: Partial Order Transparency:A Minimum Requirement for Monitoring Concurrent Systems, *Proc. 2nd International Workshop on Software Engineering and its Application*, pp. 827–839, Toulouse, France, December 1989.

[6] 程 京徳, 荒木啓二郎, 牛島和夫: Ada 並列プログラムの事象駆動型実行モニタ EDEN の開発と応用, 情報処理学会論文誌, Vol. 30, No. 1, pp. 11–24, January 1989.

[7] Cheng, J.: A Classification of Tasking Deadlocks, *ACM Ada Letters*, Vol. 10, No. 5, pp. 110–127, 1990.

[8] Cheng, J.: Task-Wait-For Graphs And Their Application To Handling Tasking Deadlocks, *Proc. ACM 3rd Annual TRI-Ada Conference*, pp. 376–390, Baltimore, USA, December 1990.

[9] Cheng, J.: A Survey of Tasking Deadlock Detection Methods, *ACM Ada Letters*, Vol. 11, No. 1, pp. 82–91, 1991.

[10] Cheng, J., Kasahara, Y. and Ushijima, K.: A Tasking Deadlock Detector for Ada Programs, *Proc. IEEE-CS 15th Annual COMPSAC*, pp. 56–63, 1991.

[11] Cheng, J. and Ushijima, K.: Partial Order Transparency as a Tool to Reduce Interference in Monitoring Concurrent Systems, in *Distributed Environments*, (Ohno, Y., ed.), pp. 156–171, Springer-Verlag, 1991.

[12] Cheng, J.: Task Dependence Net as a Representation for Concurrent Ada Programs, in *Lecture Notes in Computer Science*, Vol. 603, pp. 150–164, Springer-Verlag, 1992.

[13] Cheng, J.: The Tasking Dependence Net in Ada Software Development, *ACM Ada Letters*, Vol. 12, No. 4, pp. 24–35, 1992.

[14] Cheng, J.: Process Dependence Net of Distributed Programs and Its Applications in Development of Distributed Systems, *Proc. IEEE-CS 17th Annual COMPSAC*, pp.231–240, 1993.

[15] Cheng, J.: Complexity Metrics for Distributed Programs, *Proc. IEEE-CS 4th Annual ISSRE*, pp. 132–141, 1993.

[16] Cheng, J., Kasahara, Y., Kamachi, M., Nomura, Y. and Ushijima, K.: Compiling Programs to Their Dependence-based Representations, *Proc. 1993 IEEE TENCON*, Vol. 1, pp. 374–377, 1993.

[17] Cheng, J.: Slicing Concurrent Programs — A Graph-Theoretical Approach, in *Lecture Notes in Computer Science*, Vol. 749, pp. 223–240, Springer-Verlag, 1993.

[18] Cheng, J.: Nondeterministic Parallel Control-Flow / Definition-Use Nets and Their Applications, in *Parallel Computing: Trends and Applications*, (Joubert, G. R., *et al.*, ed.), pp. 589–592, Elsevier Science B.V., North Holland, 1994.

[19] Ferrante, J., Ottenstein, K. J. and Warren, J. D.: The Program Dependence Graph and Its Use in Optimization, *ACM Trans. Prog. Lang. Syst.*, Vol. 9, No. 3, pp.319–349, 1987.

[20] Gallagher, K. B. and Lyle, J. R.: Using Program Slicing in Software Maintenance, *IEEE Trans. Softw. Eng.*, Vol. 17, No. 8, pp. 751–761, 1991.

[21] German, S. M.: Monitoring for Deadlock and Blocking in Ada Tasking, *IEEE Trans. Softw. Eng.*, Vol. SE-10, No. 6, pp. 764–777, 1984.

[22] Horwitz, S., Reps, T. and Binkley, D.: Interprocedural Slicing using Dependence Graphs, *ACM Trans. Prog. Lang. Syst.*, Vol. 12, No. 1, pp. 26–60, 1990.

[23] Horwitz, S. and Reps, T.: The Use of Program Dependence Graphs in Software Engineering, *Proc. 14th ICSE*, pp. 392–411, 1992.

[24] Kadia, R.: Issues Encountered in Building a Flexible Software Development Environment: Lessons from the Arcadia Project, *Proc. ACM SIGSOFT 5th Symposium on Software Development Environments*, pp. 169-180, 1992.

[25] 笠原義晃: Ada タスキングデッドロックの動的検出, 九州大学工学部情報工学科卒業論文, 1991.

[26] Kasahara, Y., Cheng, J. and Ushijima, K.: A Task Dependence Net Generator for Concurrent Ada Programs, *Proc. the IPSJ & KISS Joint International Conference on Software Engineering '93*, pp. 315–322, 1993.

[27] Knapp, E.: Deadlock Detection in Distributed Databases, *ACM Computing Surveys*, Vol. 19, No. 4, pp. 303–328, 1987.

[28] Korel, B. and Laski, J.: Dynamic Program Slicing, *Inf. Process. Lett.*, Vol. 29, No. 10, pp. 155–163, 1988.

[29] Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B. R., and Wolfe, M. J.: Dependence Graphs and Compiler Optimizations, *Proc. 8th ACM Symposium on Principles of Programming Languages*, pp. 207–218, 1981.

[30] Neumann, P. G.: *Computer-Related Risks*, Addison-Wesley, 1995.

[31] 西和則: ペトリネットの編集・シミュレーションツールの開発, 九州大学工学部情報工学科卒業論文, 1995.

[32] Ottenstein, K. J. and Ottenstein, L. M.: The Program Dependence Graph in a Software Development Environment, *ACM Software Engineering Notes*, Vol. 9, No. 3, pp. 177–184, 1984.

[33] Ottenstein, K. J. and Ellcey, S. J.: Experience Compiling Fortran to Program Dependence Graphs, *Softw. Pract. Exper.*, Vol. 22, No. 1, pp. 41–62, 1992.

[34] Podgurski, A. and Clarke, L. A.: A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance, *IEEE Trans. Softw. Eng.*, Vol. 16, No. 9, pp. 965–979, 1990.

[35] Self, J.: Aflex – An Ada Lexical Analyzer Generator Version 1.1, UCI-90-18, 1990.

[36] Strelich, T.: The Software Life Cycle Support Environment (SLCSE): A Computer Based Framework for Developing Software Systems, *ACM SIGSOFT Softw. Eng. Notes*, Vol. 13, No. 5, pp. 35–44, 1988.

[37] 竹下亨: CASE 決定版, 共立出版株式会社, 1994.

[38] Taback, D., Tolani, D., and Schmalz, R. J.: Ayacc User's Manual Version 1.0, UCI-88-16, 1988.

[39] Tip, F.: A Survey of Program Slicing Techniques, Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), 1994.

[40] United States Department of Defense, *Reference Manual for Ada Programming Language*, ANSI/MIL-STD-1815A, 1983.

[41] United States Government: Information technology — Programming languages — Ada, ISO/IEC 8652:1995(E), 1994.

[42] Weiser, M.: Program Slicing, *IEEE Trans. Softw. Eng.*, Vol. SE–10, No. 4, pp. 352–357, 1984.

[43] Zhao, J., Cheng, J., and Ushijima, K.: An Evaluation of the Dependence-Based Complexity Metrics for Distributed Programs, *Proc. KISS InfoScience'93*, pp. 160–166, 1993.