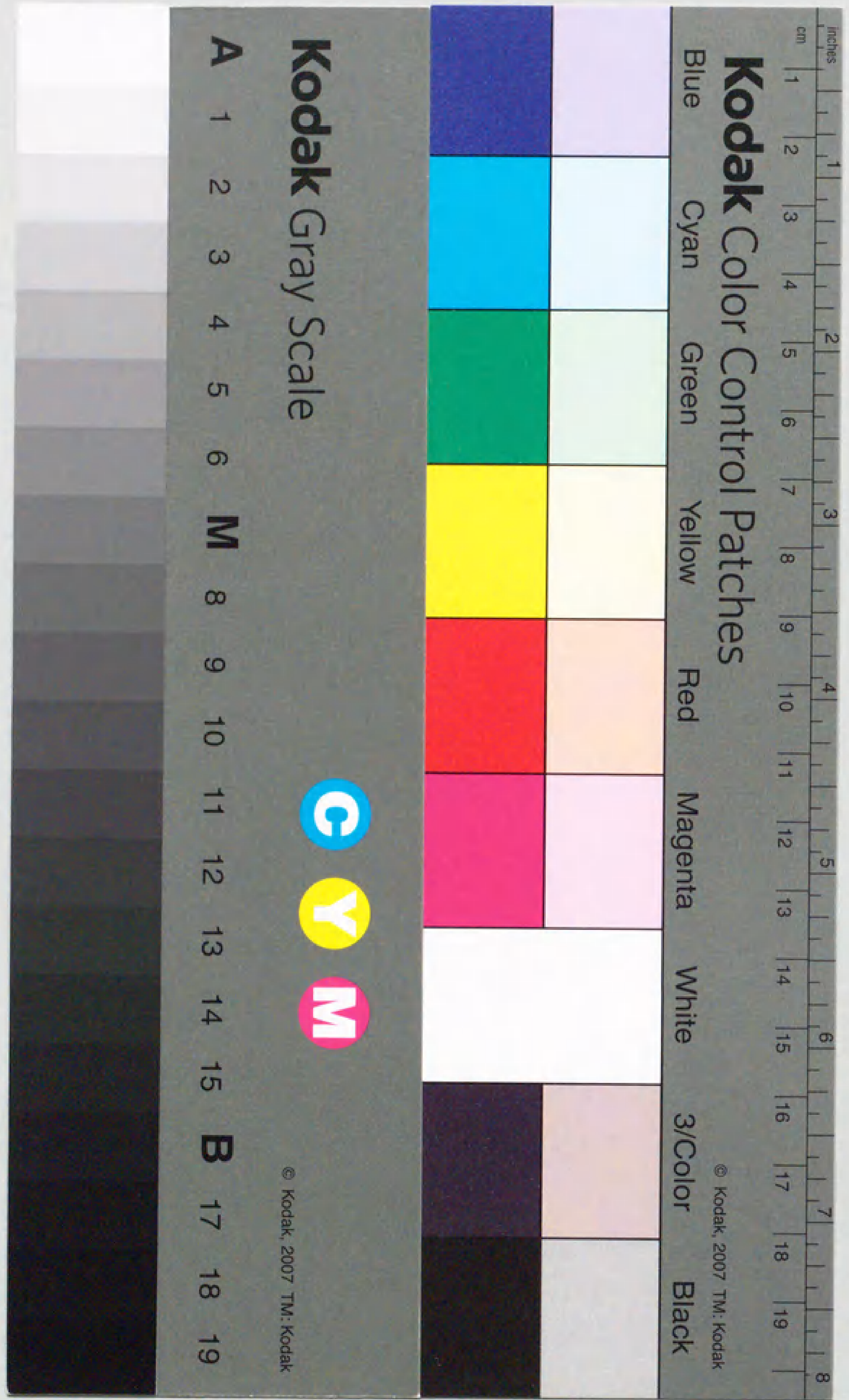


甲 4745

細粒度リポジトリに基づいたCASEツール・プラットフォームとそのソフトウェア再利用への応用に関する研究

福安 直樹

博士論文

細粒度リポジトリに基づいた CASEツール・プラットフォームと そのソフトウェア再利用への 応用に関する研究

名古屋大学大学院工学研究科
博士課程後期課程情報工学専攻

福安 直樹

概要

ソフトウェアの再利用は，システムの開発や保守にかかるコストを削減し，同時にソフトウェアの品質を向上させる．しかし，一般に既存のシステムから再利用可能な部分を見つけることは困難であり，また，見つかったとしても，通常そのままの形で再利用できることは少ないなどの問題がある．そこで，ソフトウェアの再利用を支援することが必要である．

本論文では，再利用によるソフトウェア開発の支援を目指して，上記の問題を解決する方法を示す．既存システムから再利用可能な部分を見つけ変更を支援するためには，再利用によるソフトウェア開発の各局面において様々なCASEツールを作成する必要がある．各種のCASEツール，あるいはアイデアを検証するための実験的ツールの作成にあたっては，対象言語の字句解析，構文解析，意味解析といった様々な解析を行う解析器を，ツールごとに作成する必要がある．過去に字句解析器，構文解析器の作成支援に関する多くの研究が行われているが，LEXやYACCに代表される生成系は，字句解析器，構文解析器を記述するための枠組みであって，一般に，実際のCASEツールを作成する際，即座に利用できる解析器は提供されていない．

そこで本論文では，細粒度の構成要素を対象としたCASEツール・プラットフォームを提案した．ソフトウェアライフサイクルの特に設計以降の局面では，ソースコードの字句要素や構文要素などを扱う作業が大きな割合を占める．したがって，ソフトウェアライフサイクルのあらゆる局面で生じる様々な問合せ

に応えるためには，これら細粒度の要素を扱うことが必要である．実際にC言語を対象として，提案したCASEツール・プラットフォームを実装した．我々はこれをSapid (Sophisticated Application Programming Interfaces for software Development)と呼ぶ．また，Sapidを利用していくつかのCASEツールを作成し評価を行った．これにより，提案するリポジトリが，ソフトウェアライフサイクルのあらゆる局面を支援するのに十分な粒度であり，実用的な時間的，空間的効率で動作することを示した．

次に，ソフトウェアの生産性を向上させる上で重要な技術である，ソフトウェアの再利用について議論した．ソフトウェアを再利用する際には，既存のソフトウェアを一貫性を保ちながら変更する技術が必要である．ここでは，ソースプログラムの変更に的を絞り，ユーザの視点および変更にともなう制約からソースプログラムの変更を議論した．ソースプログラムには，構文規則やスコープ規則などの様々な制約が存在する．そこで，ソフトウェアを一貫性を保ちながら変更するためのソフトウェアモデルとして，三層モデルを提案した．提案した三層モデルを用いることにより，一貫性を保ったソースプログラムの変更操作を見通し良く行うことができる．

さらに，既存システムに対して機能の追加や変更，バグの修正を行うような状況に注目し，再利用の可能な部分や，変更の必要な部分を見つける方法について考察し，既存システムを再利用することによって新システムを開発するための枠組みを提案した．また，この枠組みにしたがって新しいシステムを開発する例を示し，Sapidによって，この枠組みを支援するために必要な情報が得られることを示した．

本論文では，主にソースプログラムを対象としたが，ソースプログラムだけでなく，様々なドキュメントを統一的に管理する必要性などを今後の課題として述べた．

目次

1 序論 **1**
- 1.1 背景 1
- 1.2 目的 5
- 1.3 本論文の構成 7

2 細粒度リポジトリに基づいたCASEツール・プラットフォーム **9**
- 2.1 はじめに 9
- 2.2 細粒度リポジトリ 10
- 2.3 CASEツール・プラットフォーム 11
- 2.4 関連研究 12
- 2.5 まとめ 14

3 CASEツール・プラットフォームSapid **16**
- 3.1 はじめに 16
- 3.2 Sapid 17
- 3.3 SDB 20
- 3.4 AR 24
- 3.5 SDA 27
  - 3.5.1 制御とデータの依存関係 27
  - 3.5.2 CFG, DFGの例 30

3.6 応用と評価 30
3.6.1 ソフトウェア操作エディタ 31
3.6.2 仕様書作成ツール 33
3.6.3 Program slicing ツール 34
3.6.4 評価 36
3.7 まとめ 38

4 ソフトウェアの無矛盾な変更 39
4.1 はじめに 39
4.2 ソフトウェアの変更 40
4.2.1 ソースプログラムの変更 40
4.3 三層モデル 45
4.3.1 変更を行う際のユーザの視点 45
4.3.2 ソースプログラムの変更における制約 47
4.4 変更操作の例 49
4.4.1 コメントの追加 49
4.4.2 変数宣言の方法の統一 50
4.4.3 変数名の変更 51
4.4.4 ソースプログラムの再利用 53
4.4.5 デバッグコードの挿入 56
4.5 まとめ 61

5 再利用によるソフトウェア開発 62
5.1 はじめに 62
5.2 再利用によるソフトウェア開発の枠組み 63
5.2.1 既存システムからの要求の抽出 66

5.2.2 新システムの仕様の記述 66
5.2.3 新システムの実現 67
5.2.4 枠組みに必要な機能 67
5.3 Sapid に基づいたソフトウェア再利用 68
5.3.1 シナリオの記述 71
5.3.2 仕様の記述 72
5.3.3 機能の実現 79
5.4 再利用を支援するツール 81
5.4.1 SPIE 81
5.4.2 関数スライサ 85
5.5 まとめ 88

6 結論 90

第 1 章

序論

1.1 背景

近年，あらゆる分野において，コンピュータによる自動化が進められている．計算機の処理能力が飛躍的に向上していることから，計算機によって管理される分野は多様化している．例えば，電化製品の制御，工場の出荷管理や銀行の預金管理など，様々な規模のソフトウェアシステムが，様々な場面で利用されている．大きな規模のソフトウェアシステムを開発するためには，より多くの費用と時間が必要である．したがって，ソフトウェアの生産性向上が重要な課題である．

ソフトウェアの開発は，他の工業製品と同様の開発工程を持つ．ソフトウェアライフサイクルは，一般に要求分析から仕様記述，設計，実装，検証および検査，そして保守にいたる各段階からなっている．このようなライフサイクルの中で，ソフトウェアの開発においては，特に保守の段階が重要な役割を持つ．ソフトウェアシステムは，システムが稼働し始めてからも，そのシステムを取り巻く環境は日々変化し，システムの保守が欠かせない．また場合によっては，システムに新機能を追加したり，システムの不都合を修正し，新しいシステムを開発する必要がある．システムの規模が大きくなればなるほど，そのシステムの開発や保守にかかる費用は増大する．そこで，既存のシステムを再利用す

ることによって，新しいシステムを構築するための技術が重要となる [22].

ソフトウェアの再利用は，システムの開発や保守にかかるコストを削減し，同時にソフトウェアの品質を向上させる．既存のシステムを再利用することによって，ソースコードの記述に関してだけでなく，再利用した部分に対する分析，設計，検査の各段階においてもコストが削減される．また，実際の使用という形で十分にテストされたコードを利用することにより，ソフトウェアの信頼性が向上する．しかし，一般に既存のシステムから再利用可能な部分を見つけることは困難であり，また，見つかったとしても，通常そのままの形で再利用できることは少ないなどの問題がある．そこで，ソフトウェアの再利用を支援することが必要である．

一方，ソフトウェアの生産性向上を目指し，ソフトウェアライフサイクルの各段階を支援する目的で，様々な CASE ツールが開発されている．一般的にこれらの CASE ツールは，ソフトウェアライフサイクルを全体的に支援するものではなく，個々の段階を支援する．ソフトウェアの生産性を向上させるためには，特定の局面のみを支援するだけでは十分でないことは経験的に明らかである．すなわち，要求分析から保守にいたるすべての局面が統一的に支援されなければ，十分な効率化は達成されない．

ソフトウェアライフサイクルの各段階を支援する CASE ツールを，一つの環境の中へ統合する目的で，リポジトリによるデータ統合，トースタモデル [29] による制御統合などの各種の標準化作業が行われている [3, 26, 30]．図 1.1 にトースタモデルを示す． トースタモデルでは，各サービスは，次のようなグループに分割されている．

- オブジェクト管理
- プロセス管理

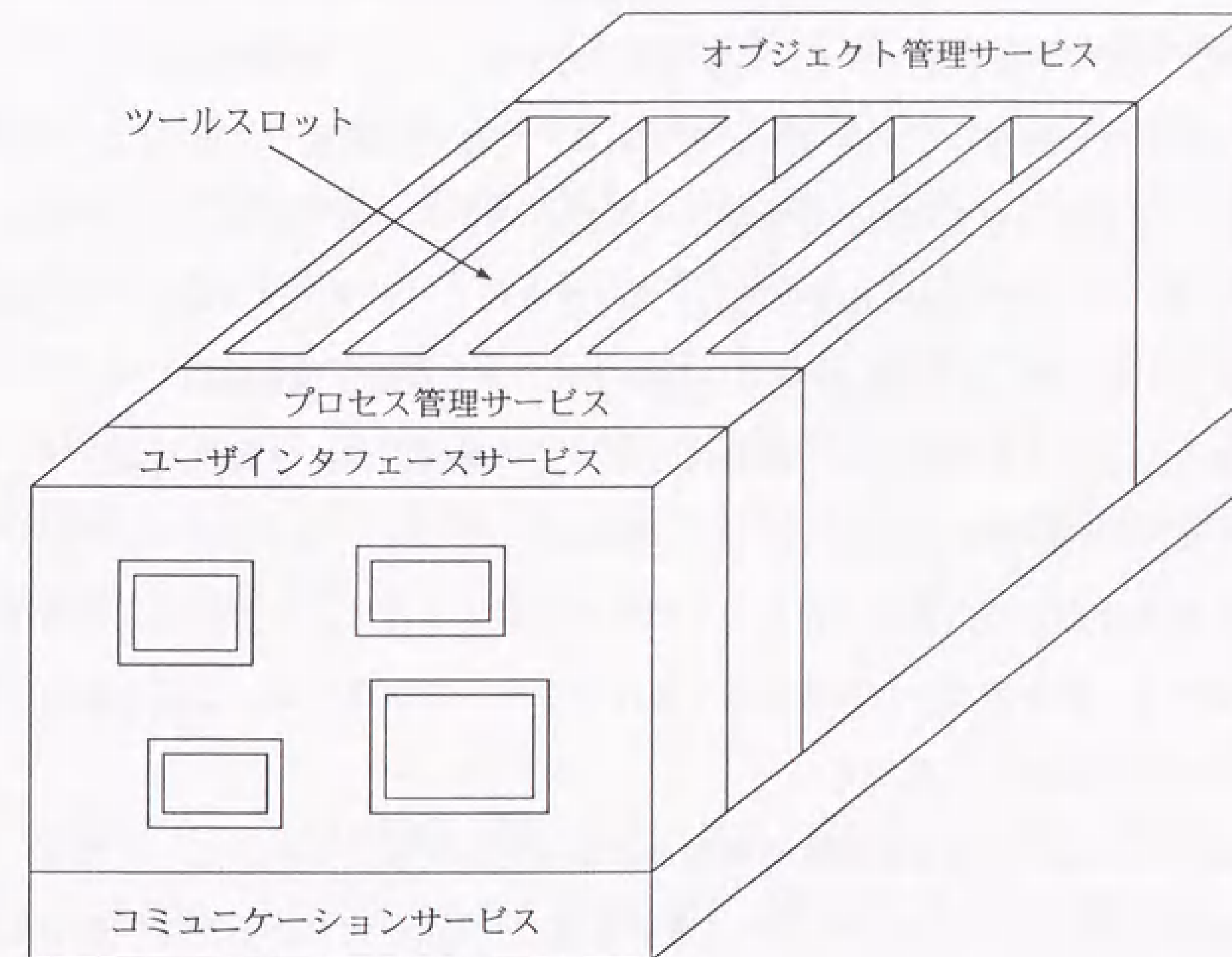


図 1.1: トースタモデル

- コミュニケーション
- ユーザインタフェース
- ツール

オブジェクト管理サービスとプロセス管理サービスは複数のツールスロットによって隔てられている．また，構造の全体は，コミュニケーションサービスによって支えられている．オブジェクト管理サービスは，データ定義や格納，データへのアクセス，操作などの各サービスである．プロセス管理サービスは，ソフトウェア開発プロセスの実行を自動化するためのサービスである．プロジェクトやリソース，スケジュールを管理し，ユーザが行うべきタスクを提示する．コミュニケーションサービスは，ツールや他のサービスに通信手段を提供する．ユーザインタフェースサービスは，環境内のすべての構成要素に一貫するユーザインタフェースを開発するためのサービスである．ツールは，特定のアプリケーションの開発や保守のための方法および技術を支援するために必要な機能を環境に付加する．他の各サービスを利用する形でツールを構築することに無駄な労力をかける必要はない [7, 33]．

これらの統合化は比較的大きな粒度の成果物を対象としている．例えば，一般的なリポジトリ [2] やプログラム理解支援ツール [4] における管理の最小単位はモジュールであり，そのモジュールの内部構造や構成要素などについての管理は行われていない．しかし，統合化の対象とならなかった細粒度の構成要素もソフトウェアシステムにおいて重要な役割を果たす．特に，ソフトウェアライフサイクルの設計以降の局面では，ソースコードの字句要素や構文要素といった細粒度の対象を扱う作業が大きな割合を占めるため，これを無視することはできない．本論文では，ソースコードの字句要素や構文要素といった，モジュールよりも粒度の細かい成果物を対象とするリポジトリを細粒度リポジトリと呼

ぶ．

各種の CASE ツール，あるいはアイデアを検証するための実験的ツールの作成にあたって，対象言語の字句解析，構文解析，意味解析といった様々な解析を行う解析器を，ツールごとに作成する必要がある．過去に字句解析器，構文解析器の作成支援に関する多くの研究が行われているが，LEX [16] や YACC [14] に代表される生成系は，字句解析器，構文解析器を記述するための枠組みであって，一般に，実際の CASE ツールを作成する際，即座に利用できる解析器は提供されていない．そのため，これらの成果を実世界のソフトウェアを対象とした CASE ツールの作成に直接使うことはできない．しかも，これらの解析器の大部分は，CASE ツールにとって不可欠ではあるが，CASE ツールの機能の本質ではないことが多い．様々な CASE ツールに共通して用いられる機能を提供する CASE ツール・プラットフォームが存在すれば，解析器を CASE ツールから分離することが可能となり，ツールの開発者はそのツールの機能の実現に専念できる．

1.2 目的

本論文では，要求分析から保守にいたるソフトウェアライフサイクルのすべての局面を統一的に支援することを目指して，細粒度の構成要素を対象とした CASE ツール・プラットフォームを提案する．

ソフトウェアライフサイクルの特に設計以降の局面では，ソースコードの字句要素や構文要素などを扱う作業が大きな割合を占める．したがって，ソフトウェアライフサイクルのあらゆる局面で生じる様々な問合せに応えるためには，これら細粒度の要素を扱うことが必要である．

実際に，C 言語を対象として，提案した CASE ツール・プラットフォームを実装する．この CASE ツール・プラットフォームを利用していくつかの CASE

ツールを作成し，提案するリポジトリが，ソフトウェアライフサイクルのあらゆる局面を支援するのに十分細粒度であることを示す．

次に，ソフトウェアの生産性を向上させる上で重要な技術である，ソフトウェアの再利用について考察する．ソフトウェアの再利用においては，一般に次のような問題点が指摘されている．

- 再利用可能なソフトウェアを発見することが困難である．
- 再利用できそうな既存のソフトウェアが見つかったとしても，そのままの形で再利用できることは少なく，変更する必要がある．

そこで，これらを支援する方法を示す．

まず，ソフトウェアを再利用する際には，既存のソフトウェアを一貫性を保ちながら変更する技術が必要である．ソフトウェアを構成するプログラムには，構文規則やスコープ規則などの様々な制約が存在する．これらの制約を満たしながら，変更操作を見通し良く行うことについて考察し，三層モデルを提案する．また，三層モデルを用いることにより，ソフトウェアに対する変更操作を見通し良く行うことが可能となることを示す．

次に，既存システムに対して機能の追加や変更，バグの修正を行うような状況に注目し，再利用の可能な部分や，変更の必要な部分を見つける方法について考察する．

UML (Unified Modeling Language) [6, 32] は，オブジェクト指向によるシステム開発で用いられる各種の図の記法を定めた言語であり，OMG (Object Management Group)[1] の標準である．近年，UML に基づいた要求分析 [11] や，仕様からオブジェクト指向言語への変換技術 [5, 27] などの研究が行われている．一般に，UML を利用したシステム開発は，ユースケース駆動によって行われる．

[1]URL: OMG: http://www.omg.org/

ユースケースは，Objectory 法 [13] において初めて導入され，UML にも採用されている．ユースケースは，システムの利用の仕方をシステム外部の視点から記述したものであり，ユースケースの具体的な事例はシナリオと呼ばれる．そこで，既存システムに対して機能の追加や変更，バグの修正を行う場合に，既存システムのシナリオを中心に仕様を記述し，この仕様を変更することによって得られた新システムの仕様との差分から，既存システムの再利用可能な部分を見つける方法について考察し，再利用によって新システムを開発するための枠組を示す．

1.3 本論文の構成

本論文では，第2章においてリポジトリの粒度について考察する．このリポジトリは，ソフトウェアのライフサイクルを支援する各種の CASE ツールが共有できる必要がある．また，各種の CASE ツールに共通して必要となる字句解析，構文解析，意味解析の機能を提供し，CASE ツールを見通し良く実現するための CASE ツール・プラットフォームについて述べる．

第3章では，UNIX を始めとする多くのシステムで利用されている C 言語を対象に，第2章で述べた細粒度リポジトリに基づいた CASE ツール・プラットフォームを実装する．我々はこの CASE ツール・プラットフォームを Sapid (Sophisticated Application Programming Interfaces for software Development) と呼ぶ．また，Sapid を用いて作成した CASE ツールを示し，Sapid を定性的かつ定量的に評価する．

第4章では，ソフトウェアの再利用に必要となる，ソフトウェアを一貫性を保ちながら変更する技術について考察する．ソフトウェアを構成するプログラムには，構文規則やスコープ規則などの様々な制約が存在するが，これらの制約を満たしながら，変更操作を見通し良く行うために三層モデルを提案する．

第5章では，既存システムに対して機能の追加や変更，バグの修正を行うような状況に注目し，既存システムを再利用することによって，新しいシステムを開発するための枠組みを提案する．この枠組の中で，Sapid の役割について考察し，枠組みの支援に必要な機能を挙げる．

最後に，第6章では，本論文の全体のまとめと考察を行う．

第 2 章

細粒度リポジトリに基づいたCASEツール・プラットフォーム

2.1 はじめに

ソフトウェアの生産性向上を目指して，様々なCASEツールが開発されている．これらのCASEツールは，ソフトウェアライフサイクルの個々の段階を支援する．ソフトウェアの生産性を向上させるためには，ソフトウェアライフサイクルの特定の局面のみを支援するだけでは十分ではないため，各種のCASEツールを，一つの環境の中へ統合する目的で，リポジトリによるデータ統合，トースタモデルによる制御統合などの各種の標準化作業が行われているが，多くの場合，管理の最小単位はモジュールである．しかし，モジュールの内部構造やモジュールを構成している要素なども，ソフトウェアシステムにおいては重要な役割を果たす．特に，ソフトウェアライフサイクルの設計以降の局面では，モジュールの内部構造や構成要素などを扱う作業が大きな割合を占める．

また，各種のCASEツールの作成には，対象言語の字句解析，構文解析，意味解析を行う解析器を作成することが必要である．従来のCASEツールにおいて，これらの解析器は各CASEツールごとに作成されていた．しかし，字句解析，構文解析，意味解析を行う解析器は，各種のCASEツールに共通であり，CASEツールにこれらの機能を提供するためのプラットフォームが望まれる．

一方，それぞれのCASEツールが必要とするデータの粒度は様々である．例えば，ソースプログラム内の識別子を置き換える場合には，スコープ規則に基づいた識別子の解析や，その出現位置などが重要となるが，program slicing [34] を行うツールでは，ソースプログラムの制御依存関係やデータ依存関係などが重要となる．各種のCASEツールを支援し，ソフトウェアのライフサイクルのいろいろな局面で要求される多様な問合せや操作に対応するためには，モジュールの内部構造や構成要素などの細粒度の要素を扱う必要がある．

以下では，まずモジュールよりも粒度の細かい要素を対象とした細粒度リポジトリについて述べる．細粒度リポジトリは，ソフトウェアを構成するあらゆる要素を統一的に管理し，それらの情報を必要に応じて容易に取り出す機能を持つ．次に，細粒度リポジトリを中核とするCASEツール・プラットフォームについて述べる．CASEツール・プラットフォームは各種のCASEツールに共通して必要な，字句解析，構文解析，意味解析の機能を提供し，これを利用することによってCASEツールを見通し良く実現することができる．

2.2 細粒度リポジトリ

CASEツールはソフトウェアライフサイクルの個々の局面を支援するが，それぞれのCASEツールが必要とするデータの粒度は様々である．システムの構成管理を行うCASEツールにおいては，システムを構成するファイルやそれらのファイルから生成される実行形式などが重要である．関数間の呼出しグラフを生成するCASEツールでは，関数の定義や関数呼出しの解析などが重要である．また，ソースプログラム内の識別子を置き換えるCASEツールでは，スコープ規則に基づいた識別子の解析や，その出現位置などが重要となる．

細粒度リポジトリは，ソフトウェアライフサイクルの各局面で行われるこのような多様な問合せや操作に対応し，CASEツールが必要に応じて必要な情報

を容易に取得できなければならない．したがって，細粒度リポジトリは次のような機能を持つ．

- ソフトウェアを構成するあらゆる要素を統一的に管理する機能
- リポジトリに記憶・管理されている情報にアクセスする機能

一般に，細粒度のモデル上に適切なビューを設定して不要な情報を隠すことは容易である．一方，細粒度のリポジトリでは，管理するオブジェクトの数が増加し，性能が低下する．そこで，上述のできるだけ細かい粒度という要請と，性能とのトレードオフから適切な粒度を設定する必要がある．

2.3 CASEツール・プラットフォーム

各種のCASEツールや，あるいはアイデアを検証するための実験的ツールを作成する場合，対象ソフトウェアの字句解析，構文解析，意味解析を行う解析器は必要不可欠である．過去に字句解析，構文解析の作成支援に関する多くの研究が行われており，LEXやYACCなどの字句解析器，構文解析器を記述するための枠組みは用意されているが，実際のCASEツールを作成する場合に，即座に利用できる解析器は提供されていない．

そこで，次のような機能を実現するためにCASEツール・プラットフォームを提案する．

1. 字句解析，構文解析などの，CASEツールにとって必要不可欠な解析機能を提供する．
2. 波及解析や抽象実行に代表されるCASEツールに関する基盤技術の実現をライブラリとして蓄積・流通・再利用するためのプラットフォームとなる．
3. 細粒度のソフトウェアモデルによるCASEツール間のデータ統合を実現する．

CASE ツール・プラットフォームと CASE ツールとの関係を図2.1 に示す．CASE ツール・プラットフォームの中核は，ソフトウェアを構成する様々な粒度の要素を統一的に管理する細粒度リポジトリである．細粒度リポジトリは，2.2 節で述べたように，ソフトウェアのライフサイクルで行われる多様な問合せや操作に対応できる必要がある．そのために，ソースコードに対する従来のリポジトリが用いるモデルよりも粒度の細かいモデルが必要である．

一方，データベースにおいては，データをユーザが利用しやすくするために，ユーザの視点に合った仮想的データベース空間としてビューを設定する機能が用意されている．細粒度リポジトリにおいても，特定の CASE ツールの立場から見ると，粒度が細かすぎることがあるため，適切なビューの役割は重要である．例えば，program slicing [34] を行う CASE ツールは，ソースプログラムをデータフローグラフおよび制御フローグラフとして捉える．同様に，McCabe によるプログラム複雑度 [18] を求める CASE ツールも，ソースプログラムを制御フローグラフとして捉える必要がある．このように，ビューはソフトウェアをある特定の CASE ツールの見方で捉えるために用いられる．

CASE ツール・プラットフォームの最上位層は，波及解析や抽象実行などの基盤技術ライブラリである．CASE ツールはこれらのライブラリを適切に組み合わせて作成される．

なお，図2.1 における各要素は必ずしも網羅的なものではない．また，第3章で説明する Sapid では C 言語を対象としているが，Sapid の基本的なアイデアは他の言語にも適用可能である [8]．

2.4 関連研究

関連研究として 2 つのシステムを挙げる．

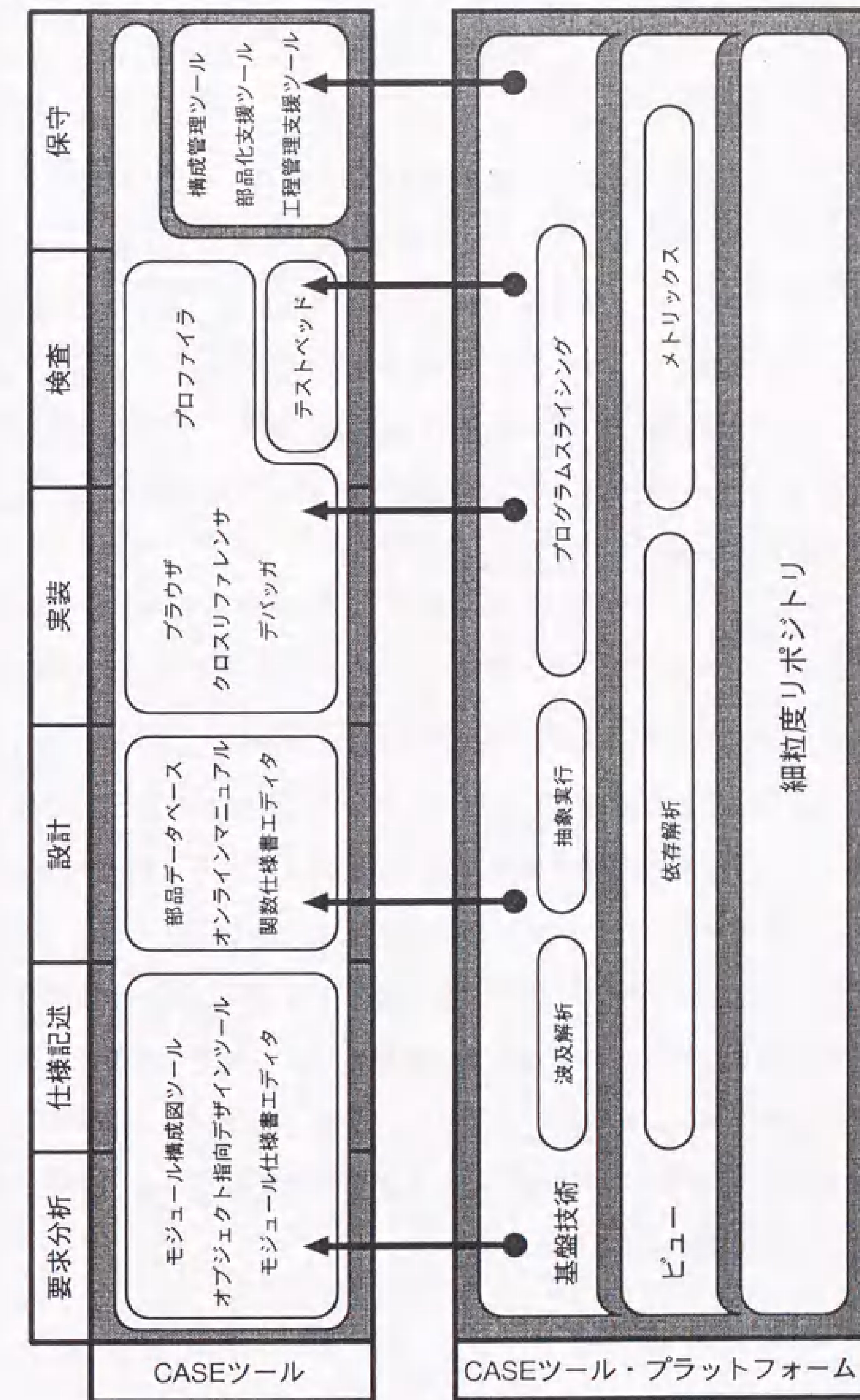


図 2.1: CASE ツール・プラットフォームの構成

COBOL アナライザ

COBOL アナライザ [15] は，商用システムの一部として用いることができる高い信頼性と性能を持っている．

文献 [15] では「構文木」としか触れられていないが， COBOL アナライザは，一般のコンパイラにおける構文木と記号表の情報を管理していると考えられる．ターゲットマシンの動作の系列をプログラムの意味とする操作的意味論の立場に立てば，十分に細かいソフトウェアのモデルを提供している．また， COBOL アナライザは制御依存関係とデータ依存関係を解析する API を提供している．

COBOL アナライザの主な目的は，対象プログラムの解析であり，変更必要箇所の発見を支援することに重点が置かれている．

REFINE

Reasoning Systems 社の Refine Language Tools [24] (以下， REFINE) は，リエンジニアリングのためのワークベンチとして有名である． REFINE は，利用者がカスタマイズ可能な強力な構文解析器を備えており，構文の定義を与えることによって種々の言語に適用できる．また， C や COBOL などの構文定義はあらかじめ用意されている． REFINE 言語インタプリタは，木や集合など様々なデータ型を持ち，それらに対し様々な演算を行うことができる．また， GUI によって抽象構文木などを表示することができる． REFINE 上で開発したプログラムは REFINE 言語インタプリタでのみ実行可能である． REFINE では依存解析などの高度な解析までは行っていない．

2.5 まとめ

本章では，特にソフトウェアライフサイクルの設計以降の局面で重要となる，モジュールよりも粒度の細かい，モジュールの内部構造やその構成要素などのデータを扱う，細粒度リポジトリについて考察した．細粒度のリポジトリでは，管理するオブジェクトの数が増加し，性能が低下するため，できるだけ細かい粒度という要請と，性能とのトレードオフから適切な粒度を設定する必要があることを述べた．

また，字句解析，構文解析などの，すでに技術的に確立されており，各種の CASE ツールで共通に必要な機能を提供し， CASE ツールを見通し良く実現するための基盤として， CASE ツール・プラットフォームを提案した． CASE ツール・プラットフォームの中核は，ソフトウェアを構成する様々な粒度の要素を統一的に管理する細粒度リポジトリである．また，細粒度リポジトリは，特定の CASE ツールにとっては，粒度が細かすぎることがあるため，適切なビューを設定する必要性について述べた．さらに， CASE ツール・プラットフォームの最上位層は，波及解析や抽象実行などの基盤技術ライブラリであり， CASE ツールはこれらのライブラリを適切に組み合わせて作成されることを述べた．

第 3 章

CASE ツール・プラットフォーム Sapid

3.1 はじめに

第2章で述べた，細粒度リポジトリに基づいた CASE ツール・プラットフォームを，UNIX を始めとする多くのシステムで利用されている C 言語を対象に実装した．我々はこの CASE ツール・プラットフォームを Sapid と呼ぶ．本章では，Sapid の構成を示し，Sapid の中核である細粒度リポジトリが用いるモデルについて説明する．次に，このモデルに基づいて解析された情報を CASE ツールが利用する方法を示す．また，依存解析を扱うためのモデルについて示す．さらに，Sapid を用いて作成した CASE ツールを示し，Sapid を定性的かつ定量的に評価する．

Sapid の目標を以下に示す．

1. 字句解析，構文解析などの，すでに技術的に確立されているが CASE ツールにとって本質でない基礎的な機能を提供する．
2. 波及解析や抽象実行に代表される CASE ツールに関する基盤技術の見通し良い実現を可能とする．また，基盤技術の実現をライブラリとして蓄積・流通・再利用するためのプラットフォームとなる．
3. 新しい CASE ツールを支える要素技術を確立するための実験基盤となる．

4. 細粒度のソフトウェアモデルによる CASE ツール間のデータ統合を実現する．

3.2 Sapid

第2章で述べた CASE ツール・プラットフォームとして Sapid を実現した．Sapid は，ソフトウェアのライフサイクルで行われる多様な問い合わせや操作に対応することのできる CASE ツール・プラットフォームを目指すため，その基盤となるリポジトリは細粒度である必要がある．

そこで，ソースコードに対する従来のリポジトリが用いるモデルよりも粒度の細かいモデル I-model を用意した．図3.1 に I-model を示す．

モデルの粒度の設定には若干の恣意性が避けられず，定量的な議論は困難であるが，以下の点から I-model は通常のプログラマにとって十分細かい粒度を持つ．I-model に基づいた解析結果を用いて C 言語の文を直接解釈実行する仮想機械を作成できることから，仮想機械の動作系列をプログラムの意味とする操作的な意味論に基づいたプログラムの等価性を議論することが可能である．また，一般に細粒度のモデル上に適切なビューを設定して不要な情報を捨象することは容易であり，粒度の粗いモデルでは将来発生する可能性がある多様な要求に応えることができない．

一方，細粒度のリポジトリでは，管理するオブジェクトの数が増加し，性能が低下する．I-model の粒度は，上述のできるだけ細かい粒度という要請と，性能のトレードオフから設定された．I-model に関する説明は 3.3 節で行う．

Sapid には，I-model 上に制御依存関係とデータ依存関係を表すビュー，C 言語を直接解釈実行する仮想機械のビューなどが階層的に用意されているため，プログラムのライフサイクルにおける中・下流工程で行われる幅広い操作に対応できる．また，利用者が簡潔にビューを記述するための枠組みとしてソフト

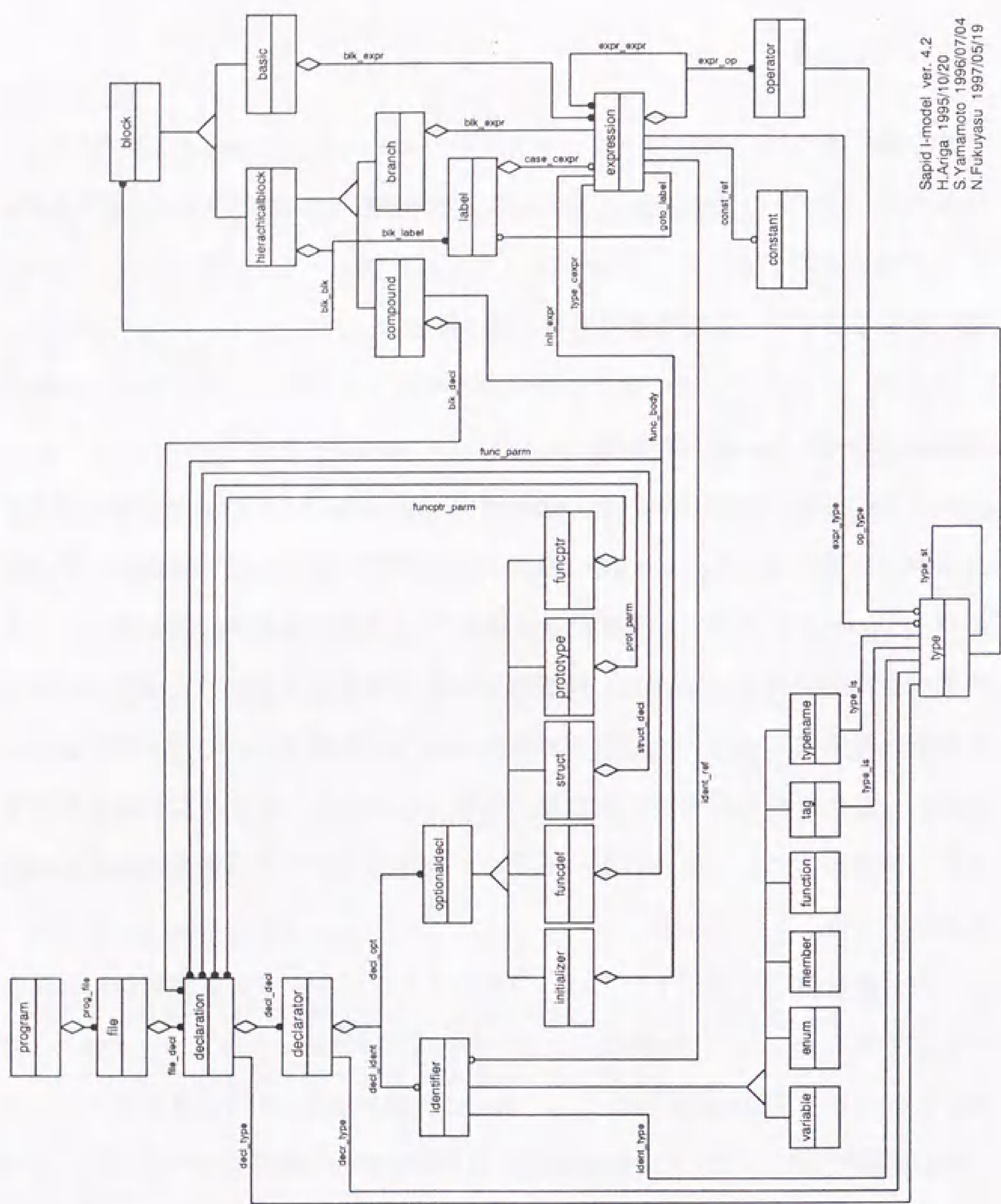


図 3.1: I-model

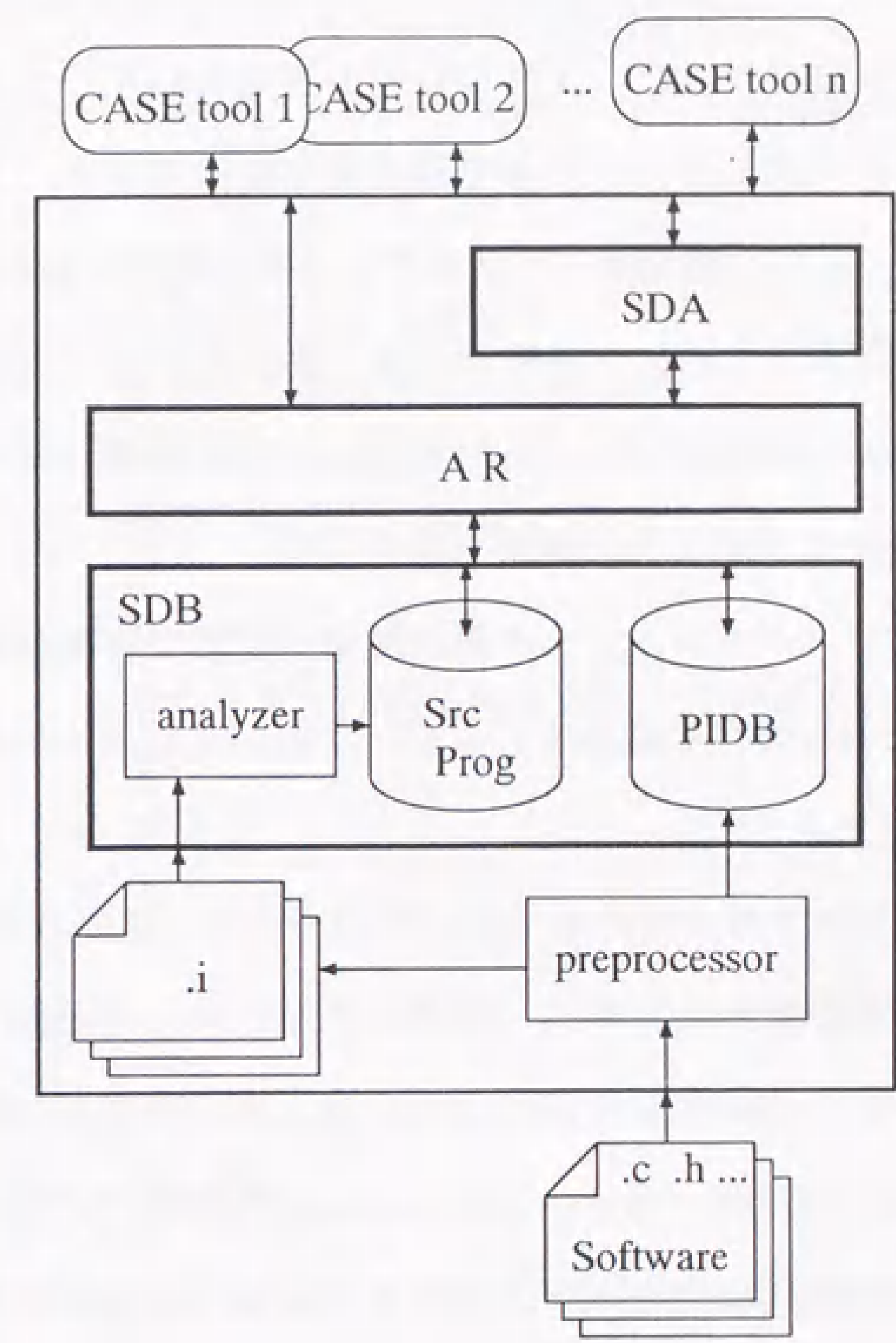


図 3.2: Sapid の構成

ウェア操作言語 [36] も用意されている．

Sapid は，ソフトウェアデータベース (SDB: Software Database) [17, 35]，アクセスルーチン (AR: Access Routines) を主要なサブシステムとする．また，ソースプログラムをデータフローグラフや制御フローグラフで捉えるためのビュー (SDA: Sapid Dependency Ananlyzer) も提供されている．システムの構成を図 3.2 に示す．

SDB は Sapid の基盤であり，要求されたソフトウェアをソフトウェアモデル I-model に基づいて解析する解析器と解析結果を格納するデータベースからなる．

C 言語のソースプログラムは，通常いくつかの前処理指令を含む．したがって，C 言語のソースプログラムに，マクロ展開や指定されたファイルの包含などの前処理を施すことによって，はじめて構文規則を適用することができる．そこで，Sapid では，入力されたソースプログラムに前処理を施してから I-model に基づいた解析を行うとともに，前処理の際に得られる，マクロ展開などの情報を，PIDB (Preprocess Information Database) [19] に蓄積する．

AR は SDB に対するアクセス機能を提供する．

SDA はソースプログラムをデータ依存関係や制御依存関係を扱うための API 関数群である．SDA を用いることにより，I-model に高い抽象度のビューを設定することができる．

Sapid は全体で約 77,000 行の C 言語プログラムから構成される．そのうち SDB は約 12,600 行，AR は約 6,400 行，PIDB は約 12,400 行 (ただし，約 6,500 行は gcc から流用)，SDA は約 5,900 行である．Sapid は現在，HP，Sun，SGI などの計算機で稼働している．また，anonymous FTP や WWW [25] 上で一般に公開し，複数の開発現場において CASE ツールの作成に用いられている．

3.3 SDB

Sapid の基盤である SDB はソフトウェアを以下に示すソフトウェアモデル I-model に基づいて解析し，その構造や様々な関連を格納する．Sapid ではソフトウェアを 12 のクラス (表 3.1)，およびそれらの間の構成関係や定義参照関係などを表す 29 の関連 (表 3.2) からなるモデル I-model (図 3.1) として捉えた．

program クラスのオブジェクトは，実行可能なプログラムを表す．file クラスのオブジェクトは，C 言語の文法規則に則して記述されているファイルを表し，複数のファイルによってプログラムを構成している．また，ファイルは一連の宣言や関数定義によって構成される．declaration クラスのオブジェクトは，型

表 3.1: I-model の 12 のクラス

program	プログラム
file	C 言語のソースファイル
declaration	変数や型などの宣言，関数定義
declarator	宣言内の個々の宣言子
identifier	変数名や関数名などの識別子
optionaldecl	関数定義の引数や本体，構造体のメンバ定義など
block	複文，選択文，繰返し文など
expression	式
label	ラベル
constant	整数定数，浮動小数点定数，文字定数，文字列定数
type	型
operator	演算子

や変数の宣言，関数のプロトタイプ宣言，関数定義，構造体のメンバ，列挙型の列挙子，および関数の引数宣言を表す．各 declaration は型との関連を持ち，複数の宣言子によって構成される．declarator クラスのオブジェクトは，宣言内の個々の宣言子を表し，その宣言子において宣言される識別子と，初期値式などによって構成される．identifier クラスのオブジェクトは，宣言子によって宣言される識別子であり，変数名や関数名，型名などを表す．識別子は式などから参照される．optionaldecl クラスのオブジェクトは，関数のプロトタイプ宣言における引数，関数定義における引数や関数本体，変数宣言における初期値式，構造体のメンバ定義，関数ポインタの宣言における引数を表す．block クラスのオブジェクトは，式文やジャンプ文の連続，複文，選択文，繰返し文を表す．expression クラスのオブジェクトは式を表し，複数の式や演算子によって構成される．また，その式が関数呼出しや変数や定数の参照の場合には，対応する識別子や定数への参照関係を持つ．label クラスのオブジェクトは，識別子ラベル，case ラベル，default ラベルを表す．constant クラスのオブジェクト

表 3.2: I-model の 29 の関連

blk_blk	block を構成している block
blk_decl	block に含まれる宣言
blk_expr	if 文などにおける条件式など
blk_label	block に含まれるラベル
case_cexpr	case ラベルに含まれる定数式
const_ref	式から参照される定数
decl_decl	宣言に含まれる宣言子
decl_ident	宣言子に含まれる識別子
decl_opt	宣言子に含まれる optionaldecl
decl_type	宣言の型
decr_type	宣言子の型
expr_expr	式を構成している式
expr_op	式で利用されている演算子
expr_type	式の型
file_decl	ファイルを構成している宣言や関数定義
func_body	関数の本体
func_parm	関数の引数
funcptr_parm	関数ポインタの引数
goto_label	goto 文が指し示すラベル
ident_ref	式から参照される識別子
ident_type	識別子の型
init_expr	初期化式
op_type	sizeof 演算子やキャストにおける型
prog_file	プログラムを構成するファイル
prot_parm	プロトタイプ宣言の引数
struct_decl	構造体のメンバの宣言
type_cexpr	配列の大きさを表す定数式
type_is	型名が表す型
type_st	型の構成

は，整数定数，浮動小数点定数，文字定数，文字列定数を表す．type クラスのオブジェクトは，関数定義や様々な宣言で利用される型を表す．operator クラスのオブジェクトは，算術演算子，関係演算子，代入演算子などの演算子を表す．

I-model は構文規則に基づいて作られたが，ソフトウェア開発者の立場から必要となるクラスや関連を加えたり，逆に重要ではないものを削除することで，モデルを単純で直観的なものにした．なお，ソフトウェアの構成要素には，仕様書，構成管理書類などの各種ドキュメントがあるが，現段階ではソースプログラムに重点を置いているため，ソースプログラムに関係しないクラスと関連は省略している．

関数，変数，型，定数，構造体のメンバはクラスとしてモデル化されているため，名前の置換えや参照情報の抽出は容易である．例えば，有効範囲や名前空間に基づいて，大域変数，局所変数，構造体のメンバなどを整合性を保ちつつ置換することができる．

C 言語を用いたソフトウェアの開発で多く用いられる前処理に対処するために，I-model に基づいた解析が行われる前に，マクロ展開や指定されたファイルの包含などの前処理が施される．この前処理の際のマクロ展開などの情報は，前処理情報データベース PIDB に蓄積される．前処理系が持つマクロの展開や条件付コンパイルなどの機能は，ソフトウェアの柔軟な開発・保守・管理に有用である．しかし，ソースプログラムに含まれる前処理の対象となる記述はその言語の構文規則に従わない場合もあり，解析の障害になることが多い．I-model に前処理に関する情報を持たせると，I-model が複雑になるため，I-model は前処理後のソースプログラムに関する情報のみを格納している．その代わり，前処理の際に行われたマクロ展開などの情報を PIDB に格納する．さらに，PIDB と I-model に基づいた解析の結果から前処理する前のソースプログラムと I-model

によって解析されたソースプログラムの対応関係についても SDB に蓄積する．ユーザは 3.4 節で述べる AR を利用してこれらの情報に容易にアクセスすることができる．

3.4 AR

AR は SDB にアクセスするための API 関数群である．AR で用意されている関数は，CASE ツール作成者にとって必要かつ基本的なものである．API として，データベーススキーマを得るためのメタデータ取得関数，オブジェクトや関連の属性値取得関数，オブジェクト取得関数，関連取得関数などが用意されている．メタデータとは，SDB 内のデータを構成するクラス名，関連名，およびクラスと関連の持つ属性名を表す．以下に，API として用意されている関数の一部を示す．

メタデータ取得関数　SDB 内のデータを構成するクラス名，関連名，およびクラスと関連の持つ属性名とその識別番号の対応を得る関数である．メタデータ取得関数として，次のような関数が用意されている．

- `spdGetClassNameById(classId);`
  クラス・関連の識別番号 (`classId`) を与えると，対応するクラス名・関連名を返す．

- `spdGetClassIdByName(className);`
  クラス名・関連名 (`className`) を与えると，対応するクラス・関連の識別番号を返す．

- `spdGetAttrIdByName(className, attrName);`
  クラス名・関連名 (`className`) と，属性名 (`attrName`) を与えると，その属性の識別番号を返す．

初期化関数　指定したモデルに基づいて SDB を読み込むための関数である．

- `spdTarget(modelName, programName);`
  `modelName` で示されるモデルに基づいて SDB を読み込み，プログラム名が `programName` のプログラムの識別番号を取得する．

属性値取得関数　オブジェクトや関連の属性値を取得するための関数である．次のような関数が用意されている．

- `spdGetName(objectId);`
  識別番号が `objectId` であるオブジェクトの属性 `name` の値を返す．program や file，identifier クラスなどのオブジェクトが属性 `name` を持つ．

- `spdGetLength(objectId);`
  識別番号が `objectId` であるオブジェクトの属性 `length` の値を返す．属性 `length` は，オブジェクトの長さを表す．

- `spdGetOffset(relationId);`
  識別番号が `relationId` である関連の属性 `offset` の値を返す．関連の属性 `offset` は，出現位置をファイルの先頭からの文字数で表す．

- `spdGetAttrValInt(objectId, attrName);`
  `spdGetAttrValString(objectId, attrName);`
  識別番号が `objectId` であるオブジェクトが持つ属性のうち，属性名が `attrName` である属性の値を返す．`spdGetAttrValInt()` は属性値が整数の場合に，`spdGetAttrValString()` は属性値が文字列の場合に使用される．

オブジェクト取得関数　指定したクラスに所属するオブジェクトや，指定した関連によって関連付けられているオブジェクトを取得する関数である．

- `spdGetObjIdInit(className, name);`
  `spdGetObjId(cursor);`
  `spdGetAnObjId(className, name);`
  クラス名が className であるクラスに所属し，属性 name の値が name のオブジェクトの識別番号を取得する．spdGetObjIdInit() によって条件の設定を行い，その返り値を spdGetObjId() に与えることによって，条件を満たすオブジェクトの識別番号を次々に得ることができる．name に NULL を指定した場合には，className で指定されたクラスに所属する全てのオブジェクトの識別番号が得られる．また，spdGetAnObjId() は条件を満たすオブジェクトを一つだけ得るための関数である．

- `spdGetRelObjInit(objectId, relationName, attrName);`
  `spdGetRelObj(cursor);`
  `spdGetARelObj(objectId, relationName, attrName);`
  関連 relationName の属性 attrName が，識別番号 objectId のオブジェクトである場合に，その関連によって関連付けられているオブジェクトの識別番号を返す．

関連取得関数　指定したオブジェクトに関する関連を取得する関数である．

- `spdGetRelInit(objectId, relationName, attrName);`
  `spdGetRel(cursor);`
  `spdGetARel(objectId, relationName, attrName);`
  関連 relationName の属性 attrName が，識別番号 objectId のオブジェクトである場合に，その関連の識別番号とその関連によって関連付けられているオブジェクトの識別番号の組を返す．

- `spdGetRelIdInit(objectId1, objectId2, relationName);`
  `spdGetRelId(cursor);`
  `spdGetARelId(objectId1, objectId2, relationName);`
  関連 relationName に所属する関連のうち，オブジェクト objectId1 と objectId2 を関連付けている関連の識別番号を返す．

その他の関数

- `spdFreeCursor(cursor);`
  spdGetObjIdInit() などの関数が，指定した条件による検索のために確保した領域を解放する．

Sapid を用いる CASE ツールは，これらの関数を組み合わせて，より複雑な機能を作成する．AR を用いると，

- 構造体のメンバ foo に代入している箇所とその値を参照している箇所をあげよ (図 3.3)
- 関数 printf() の呼出しで，第 1 引数が文字列定数 "Hello" であるものをあげよ (図 3.4)

などの操作を容易に実現することができる．

3.5 SDA

3.5.1 制御とデータの依存関係

SDA は，ソースプログラムの制御依存関係とデータ依存関係を扱う CASE ツールを見通し良く実現するためのライブラリである．SDA を用いることにより，I-model に高い抽象度のビューを設定することができる．

```
 1  SpdCursor *csr;
 2  SpdObjId expr_id, ident_id;
 3  SpdRel ident_ref;
 4
 5  /* すべての式について， */
 6  csr = spdGetObjIdInit("expression", NULL);
 7  while ((expr_id = spdGetObjId(csr)) != SAPID_NON_ID) {
 8    ident_ref = spdGetARel(expr_id, "ident_ref", "expression_id");
 9    ident_id = ident_ref.objectId;
10
11    /*
12     * 関連 ident_ref によって関連付けられる識別子が存在し，
13     * その識別子が構造体のメンバで，名前が "foo" ならば，
14     * その式の識別番号を出力する．
15     */
16    if (ident_id != SAPID_NON_ID &&
17        spdGetAttrValInt(ident_id, "sort") == ID_MEMBER &&
18        SAPID_STREQ(spdGetName(ident_id), "foo"))
19      printf("%d\n", expr_id);
20  }
21  spdFreeCursor(csr);
```

図 3.3: 構造体のメンバ foo に代入している箇所とその値を参照している箇所を出力する記述例


```
 1  SpdCursor *e_csr, *a_csr;
 2  SpdObjId e_id, f_id, c_id;
 3  SpdRel arg;
 4  SpdRel *a_buf;
 5  int a_num;
 6
 7  e_csr = spdGetObjIdInit("expression", NULL);
 8  while ((e_id = spdGetObjId(e_csr)) != SAPID_NON_ID)
 9    /* すべての関数呼出し式について， */
10    if (spdGetAttrValInt(e_id, "sort") == EXPR_SORT_FUNCCALL) {
11      a_num = 0;
12      a_buf = NULL;
13      a_csr = spdGetRelInit(e_id, "expr_expr", "outer_expression_id");
14      while ((arg = spdGetRel(a_csr)).objectId != SAPID_NON_ID) {
15        a_buf = (SpdRel *)spdRealloc(a_buf,
16                                     sizeof(SpdRel) * (a_num + 1));
17        a_buf[a_num] = arg;
18        a_num++;
19      }
20      spdFreeCursor(a_csr);
21
22      if (a_num > 1) {
23        SAPID_QSORT(a_buf, a_num, sizeof(SpdRel), cmp_offset);
24        f_id = spdGetARelObj(a_buf[0].objectId,
25                             "ident_ref", "expression_id");
26        c_id = spdGetARelObj(a_buf[1].objectId, "const_ref",
27                             "expression_id");
28        /*
29         * 呼び出された関数が "printf" で，その第一引数が
30         * 文字列定数 "Hello" ならば，その関数呼出し式の
31         * 識別番号を出力する．
32         */
33        if (f_id != SAPID_NON_ID && c_id != SAPID_NON_ID &&
34            SAPID_STREQ(spdGetName(f_id), "printf") &&
35            SAPID_STREQ(spdGetAttrValString(c_id, "value"),
36                        "\"Hello\""))
37          printf("%d\n", e_id);
38      }
39      if (a_buf != NULL) spdFree(a_buf);
40    }
41  spdFreeCursor(e_csr);
```

図 3.4: 関数 printf() の呼出しで，第 1 引数が文字列定数 "Hello"であるものを出力する記述例

制御依存関係は，制御フローグラフ (CFG: Control Flow Graph) で表され，プログラムを構成するステートメントを節点とし，また 2 節点間の制御依存関係を，2 つの節点を結ぶ辺とする．

データ依存関係は，データフローグラフ (DFG: Data Flow Graph) で表され，変数の出現を節点とし，データ依存関係を辺とする．データ依存関係の定義はその目的によって様々に変わりうるが，定義 - 使用の関係など一般的な関係は，あらかじめ提供される．また，CASE ツール作成者が独自の依存関係を手続き的に与えられる仕組みも用意されている．

現在の SDA は関数を単位とした解析を行う．関数に跨った解析と alias 解析の機能の充実は今後の課題である．

3.5.2 CFG, DFG の例

図 3.5 に示したプログラムに対する CFG，DFG の例を図 3.6 に示す．図 3.6 の左側が CFG で，右側が DFG である．ただし，DFG の辺は定義 - 使用関係を表している．また，DFG の節点に対応する変数の出現は，左側の CFG の節点における変数の出現に対応している．

5 行目の `a = 1` では，変数 a が定義されている．この定義が使用されるのは，11，13 行目の `printf(...)` 文である．また，スコープの異なる a が 8 行目で定義され，9 行目で使用される．さらに，14，17 行目で定義された変数 a が 19 行目で使用される．SDA によりこれらのグラフを簡単に作成することができる．

3.6 応用と評価

Sapid を利用した CASE ツールの例として，ソフトウェア操作エディタ，仕様書作成ツール，program slicing ツールを示す．この他にも，我々の研究グループによる依存関係が定義可能なテストベッド [31]，依存関係に基づく差分抽出ツー


```
 1 void function(void)
 2 {
 3     int a;
 4
 5     a = 1;
 6     {
 7         int a;
 8         a = 22;
 9         printf("1: a = %d \n", a);
10     }
11     printf("2: a = %d \n", a);
12     if (condition()) {
13         printf("3: a = %d \n", a);
14         a = 333;
15     }
16     else {
17         a = 444;
18     }
19     printf("4: a = %d \n", a);
20 }
```

図 3.5: 被解析プログラム

ル [37] や，京都大学の小田ら [20] による C 言語を対象としたリバースエンジニアリングツールなどが Sapid を用いて作成されている．

3.6.1 ソフトウェア操作エディタ

ソースプログラム内の識別子を置き換える場合に，単なる文字列の置換えでは不十分なことは明らかである．一方，エディタに，構文に関する知識を持たせた構文エディタが提案されているが，構文規則に拘束された操作しか行えないため，あまり普及していない．我々は，自由度を制限せずに，必要なときにのみ構文を理解して置換を行うことを目的として Emacs 上に SDB と AR を用いた識別子置換コマンドを作成した [1]．

このコマンドでは，Emacs 上で任意の識別子の名前を指定し，新しい名前を入力すると，SDB の情報を基にして置換が必要な識別子をすべて置き換える．

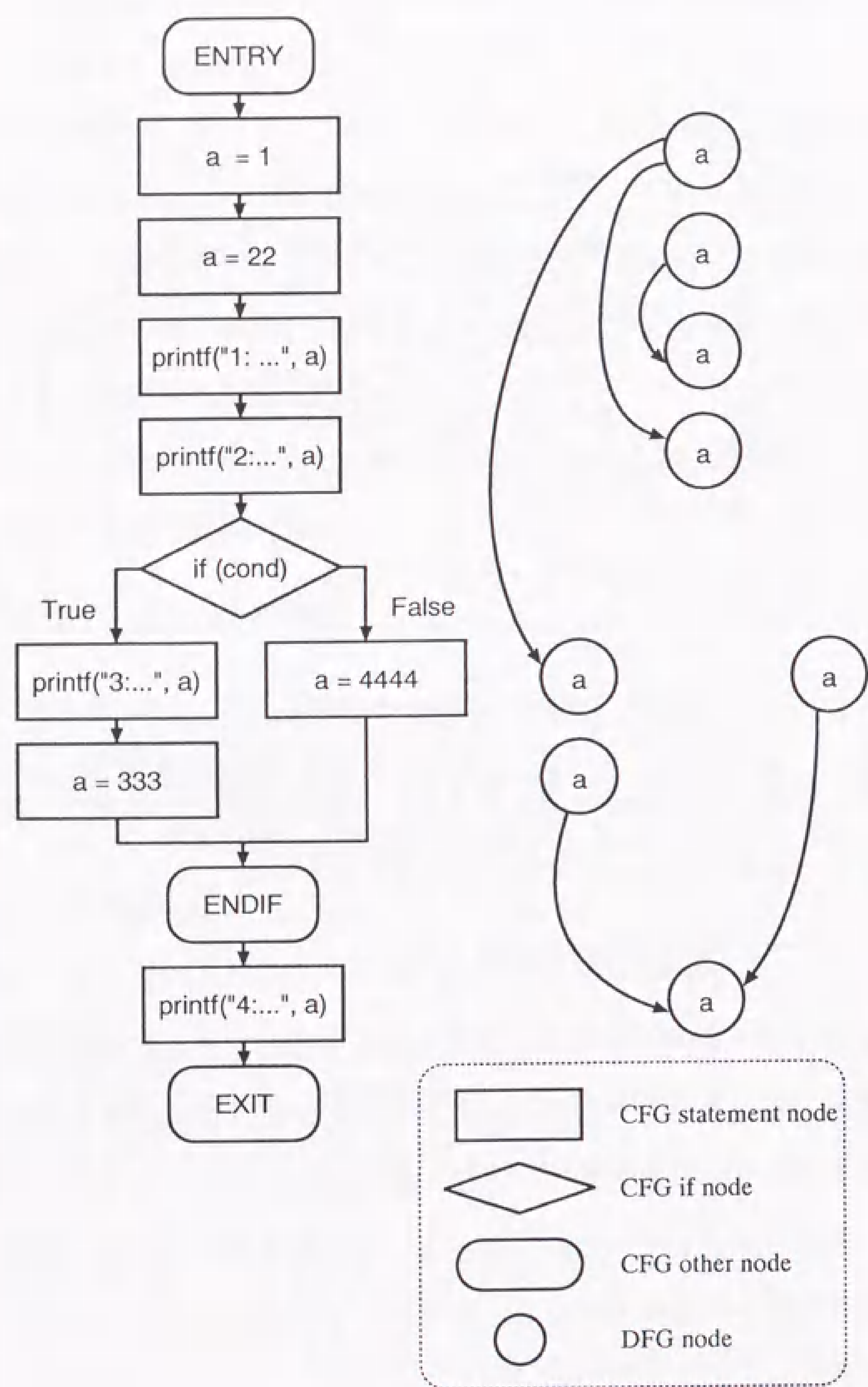


図 3.6: CFG，DFG の例

このとき，有効範囲および名前空間が異なる識別子は置き換えず，また置き換え後に名前が重複する場合にはユーザに確認するなど整合性を保つように動作する．

具体的な動作の流れは次の通りである．ユーザは，Emacs 上で置換したい識別子の一つにカーソルを移動させてから置換コマンドを実行する．置換コマンドは，カーソルの位置を取得し SDB にその識別子の出現箇所を問い合わせる．次に，ユーザに新しい名前を入力させ，その名前がすでに存在する識別子と重複するかどうかを SDB に問い合わる．重複していた場合はユーザに実行の確認を行う．整合性について問題がなければ，出現箇所にある名前をすべて置き換える．このコマンドでは，SDB に対する問合せに AR を用いている．

プログラムは，マクロの処理に関する部分も含めて emacs lisp で約 600 行である．

3.6.2 仕様書作成ツール

一般に，仕様書やソースプログラムなどの各種ドキュメントは相互に整合性が保たれなければならない．例えば，ソースプログラム上で，ある関数 f の引数の型や返り値の型が変更された場合，関数 f の仕様書や，マニュアルなどの各種ドキュメントについても同時に変更される必要がある．しかし，各種ドキュメント間の整合性を保つ作業には非常に大きな労力がかかるため，これの自動化，あるいは整合性チェックの自動化が望まれている．仕様書作成ツールは，ソースプログラムが SDB に格納されているときに，そのソースプログラムの中で定義されている関数に関して，関数仕様書の雛型を作成することができる．この生成された雛型に必要な情報を書き込むことで，仕様書を完成させることができる．

関数仕様書は，関数名とその型，関数の引数名とその型，関数内で用いられ

ている変数名とその型，呼出し関数名とその型，作成者氏名，作成年月日，最終更新年月日の 7 項目からなる．また，関数仕様書の記述には我々が作成した LaTeX のスタイルファイルを用いる．このシステムは，約 2,000 行の C 言語プログラムからなる．

Sapid の解析器 sdbin の中で定義されている関数 `analyze()` に対する関数仕様書の雛型を，この仕様書作成ツールを用いて作成した．その結果を図 3.7 に示す．

sdbin の中で定義されている関数 `analyze()` は，解析器の中心となる関数である．図 3.7 によると，関数 `analyze()` は `basic_blk`，`block_level` を始めとする多くの大域変数や `NON`，`SCOPE_FILE`，`NULL` といったマクロを使用している．渡される引数は `filename` 一つであり，値は返さない．また，関数 `analyze()` は `main()` から呼び出され，内部では構文解析を行う `yyparse()` といった関数を呼び出している．

3.6.3 Program slicing ツール

ソフトウェアの再利用においては，実行効率や可読性の向上のため，既存のソースプログラムから必要な部分のみを抽出する作業が頻繁に行われる． Weiser によって提案された program slicing[34] は，変数の定義参照の依存関係を利用して，プログラム中の指定された文に影響を与えるすべての文を抽出する技術である． program slicing の応用として，プログラムのデバッグ，メインテナンス，プログラムの理解支援などが考えられている．

我々は program slicing がプログラムのカスタマイズにどの程度利用できるかを確認するために，制限された C 言語 (ポインタを含む) を対象にして， program slicing ツールを試作した [10]．試作したツールは，制限した C 言語 (ポインタを含む) のソースプログラムのある文 s 中の変数 v を指定して，ソースプログラム

<table>
<tr><td rowspan="4">関数
analyze :void</td><td>ファイル</td><td>y.tab.c</td></tr>
<tr><td>作成者</td><td>FUKUYASU Naoki</td></tr>
<tr><td>作成年月日</td><td>Sun Dec 12 03:20:38 1999</td></tr>
<tr><td>登録年月日</td><td>Sun Dec 12 03:20:38 1999</td></tr>
<tr><td>引数
filename :char *</td><td colspan="2">グローバル変数
current_file : FOP
direct_decl_stack : struct Stack_Func *
file_decl : DO_ListP
file_func : FunctionListP
func_end_ln : int
func_parm_stack : struct Stack_Parm *
ident_ref : IDO_ListP
label_table : LO_ListP
ptr_ts_stack : struct Stack_Type *
ptr_ts_struc : TSP
scope_list : ScopeListP
source_lineno : int
specifier_flag : int
storage_stack : struct Stack_Storage *
tmp_fo : FOP
type_ref : TypeListP
yylineno : int</td></tr>
<tr><td>ローカル変数
c_count : int</td><td colspan="2">マクロ
NULL
NON
SCOPE_FILE</td></tr>
<tr><td colspan="3">関数呼出し
close_block(void) : void
escape_level(void) : void
insert_file(char *name) : FOP
insert_level(int type) : void
insert_scope_list(ScopeListP scope_list, SCP sc) : ScopeListP
pop_ts(void) : struct Stack_Type *
push_decl_decl(void) : struct Stack_Decl *
push_direct_decl(int direct_decl) : struct Stack_Func *
push_func_parm(void) : struct Stack_Parm *
push_storage(void) : struct Stack_Storage *
push_ts(void) : struct Stack_Type *
yyparse(void) : int</td></tr>
<tr><td colspan="3">呼出し元
main(int argc, char *argv[]) : int</td></tr>
<tr><td colspan="3">コメント</td></tr>
</table>

図 3.7: 関数 `analyze()` に対する関数仕様書の雛型

中の v の値に影響を与えるすべての文を抽出するツールである．このツールは約 2,000 行の C 言語のプログラムである．その他の仕様を以下に示す．

- if-then-else 文，および while 文を持つプログラムから実行可能な必要部分を抽出する．switch-case 文，goto 文，for 文は扱わない．
- 高次ポインタの解析，および関数間に跨ったポインタ型引数の解析は行わない．

X Window System 上で動作するアプリケーションの一部をなす 100 行程度の関数を対象に評価を行った．この関数は数値計算部と計算された数値の表示部からなるが，試作した program slicing ツールによって数値計算部のみを取り出す試行を行ったところ，人間が手動で抽出したものとほぼ同じ結果が得られた．

3.6.4 評価

時間的・空間的効率

X11R6.3 で配布されている端末エミュレータ xterm は 15 個の C 言語プログラムファイルからなり，合計行数は約 21,000 行である．このプログラムを I-model に基づいて解析すると，99,000 個のオブジェクトと 183,000 個の関連が生成される．そのための時間は DELL 社の Precision 410，FreeBSD-3.3 で 70 秒である．ちなみに，このプログラムに前処理を施すと全体で約 257,000 行になり，I-model はこの前処理後のソースプログラムに対して生成される．また，xterm を最適化しないでコンパイルするのにかかる時間は 7 秒である．

xterm を構成する 344 個の関数に対して，3.6.2 節の仕様書管理ツールで関数仕様書の雛型を作成するのに 27 秒かかった．出力は約 7,400 行の $\LaTeX$ ソースである．

表 3.3: CASE ツールのプログラムサイズ (単位: 行)

	Sapid 使用	Sapid 未使用
置換コマンド	800	2,400
program slicing ツール	2,000	5,000

これらの経験より，Sapid が実用的な時間的，空間的効率で動作することが分かる．

CASE ツールのプログラムサイズ

ここでは，3.6.1 節のソフトウェア操作エディタの置換コマンドと 3.6.3 節の program slicing ツールを対象にして，Sapid を用いる場合と用いない場合のそれぞれについて，CASE ツールのプログラムサイズを考察する．ただし，実際に Sapid を用いないでこれらの CASE ツールを作成するのは非現実的であるので，これらの CASE ツールが Sapid のどの部分を用いているかを分析し，その効果を以下のように推定する．

表3.3 より，Sapid を用いない場合は，Sapid を用いる場合に比べて，2.5 倍から 3 倍程度のプログラムを作成する必要があることが分かる．これらの CASE ツールは Sapid の比較的低レベルの API のみを使っているので，より高機能な API を用いた CASE ツールでは，その差はさらに大きくなると考えられる．

また，2.4 節の REFINE を利用した場合の評価に関して，REFINE を用いない場合は，REFINE を用いる場合の約 2.7 倍程度のプログラムの記述が必要であることが文献 [28] において報告されている．

評価のまとめ

3.1 節であげた Sapid の目標 (1)「CASE ツールの基礎的な機能の提供」は達成することができた．目標 (2)「CASE ツール・プラットフォームの実現」は部分的に達成することができた．今後は，このプラットフォーム上にライフサイクルのより広い範囲を支援する高機能な CASE ツールを作成し，枠組みを洗練するとともにその実用性を検証することが望まれる．目標 (3)「CASE ツールの新しい要素技術」については，文献 [20] などの若干の成果が生み出されつつあるが，Sapid を用いた革新的な技術の誕生が待たれる．目標 (4)「Sapid を用いたデータ統合」は今後の課題として残されている．

3.7 まとめ

本章では，ソフトウェアに対する変更や情報取得などの操作を行うツールを容易に構築する CASE ツールプラットフォーム Sapid を提案した．Sapid は主に SDB，AR，SDA から構成されており，SDB はソフトウェアに関する情報を保持するデータベースであり，AR は SDB にアクセスするためのルーチンである．SDB は前処理に関する情報を保持した PIDB を含み，ユーザは前処理が行われる前のソースファイルについて操作を行うことができる．SDA は，ソースプログラムをデータフローグラフや制御フローグラフで捉えるためのビューである．

また，ツールプラットフォームの応用例としてソフトウェア操作エディタ，関数仕様書管理ツール，program slicing ツールを示した．Sapid を用いることで CASE ツールのプログラムをアルゴリズムに忠実に見通し良く記述することができた．これらの経験より，Sapid が提供する API を用いて各種の中流・下流 CASE ツールを見通し良く作成できることを，定性的かつ定量的に確認した．

第 4 章

ソフトウェアの無矛盾な変更

4.1 はじめに

ソフトウェアの保守作業はソフトウェアのライフサイクルの中で重要な位置を占めるがこの作業には多大な労力を必要とする．このため，依存解析などの様々な解析の結果を利用するなどして，これらの作業を支援し，効率的に行う必要がある．

そこで，リポジトリを用いてソフトウェアを安全に変更することにより，ソフトウェアの保守や部品化，再利用などを行うことが望まれている．ソフトウェアを変更する場合，変更後も，構文規則や意味規則などの制約に関して，一貫性が保たれている必要がある．

一方第 3 章で述べたように，モジュールよりも粒度の細かい成果物を対象とする，細粒度リポジトリに基づいた CASE ツール・プラットフォームとして Sapid を提案し実現している．Sapid の基盤となっている細粒度リポジトリのモデル I-model は，ソフトウェアから得られる情報の取得を中心にモデル化されている．したがって，ソフトウェアの保守や部品化，再利用といったリポジトリの内容を変更するようなツールの開発を支援するためには，構文規則や意味規則などの制約に関して一貫性を保ちながら変更するための高度なビューが必要である．

そこで本章では，ソフトウェアの変更をユーザの視点および変更の際の制約

から考察することにより，ソフトウェアの保守や部品化，再利用などの作業を支援し，ソフトウェアの開発効率の向上を目的として，ソフトウェアの一貫性を保った変更を目指したソフトウェアモデル「三層モデル」を提案する．三層モデルはソフトウェアを表現層，構文層，記号層の三層によって表現するモデルである．

この三層モデルを用いることにより，ソフトウェアに対する変更操作を見通しよく行うことが可能となる．また，これにより再解析することなくモデル上でソフトウェアを管理することができるため，ソフトウェアの再利用や保守を効率良く行うために有効である．

4.2 ソフトウェアの変更

本研究では，変更の対象としてソースプログラムを考える．本来ならば，ソースプログラムだけでなく，関連する関数仕様書やマニュアルといった様々なドキュメントについても考慮する必要がある．しかし，ソースプログラム以外のドキュメントは各開発現場ごとに書式が異なり，これらを統一的に扱うことは難しい．ただし，ソースプログラム以外のドキュメントに関しても，書式が明確に記述されていれば，ソースプログラムと同様の手法を適用することは可能である．また，本論文では対象言語として UNIX などで広く利用されている C 言語を扱う．ただし，同様の手法は他の言語にも適用可能である．

4.2.1 ソースプログラムの変更

ソースプログラムの変更の例として次のようなものが考えられる．

- プログラムを理解しやすくするためにソースプログラムにコメントを追加する (図 4.1)
- 昇順のソートから降順のソートに変更するために判定式を変更する (図 4.2)


```
fp = fopen(filename, "r");
if (fp == NULL) {   /* file not exist */
    fprintf(stderr, "Can't open file %s.",
            filename);
    exit(1);
}
...
```

図 4.1: コメントを追加する変更

- プログラムにデバッグコードと必要な宣言を追加する (図 4.3)

これらの変更は，通常エディタを用いて行われるが，人間がこのような作業を行う際には，様々な制約に関する知識を利用している．

まず，コメントを追加する場合には，コメントだけを追加するだけでなく，コメントをわかりやすい位置に付け，また読みやすいように適切に空白や改行を追加する．C 言語ではコメントが空白や改行と同様に扱われ，これらが字句解析の段階で捨てられるので，構文解析には影響しない．

式の変更やデバッグコードの追加の場合には，構文を意識して文単位や式単位の変更が行われる．

宣言を追加する場合には，宣言が行われる同一のスコープに同名の識別子がないことを意識して行われる．

ソースプログラムに対する変更は以下の三つに分類することができる．

1. 構文に影響を与えない変更
2. 識別子の宣言に影響を与えない構文の変更

```
for (i = 0; i < n; i++) {
    for (j = n - 1; j > i; j--) {
        if (a[j-1] > a[j]) {
            int tmp;
            tmp = a[j-1]; a[j-1] = a[j]; a[j] = tmp;
        }
    }
}
```

図 4.2: 条件判定式の変更

3. 識別子の宣言に影響を与える変更

以下ではそれぞれの変更について詳しく見ていくことにする.

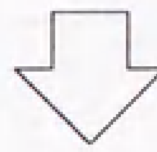
構文に影響を与えない変更

コメントの追加や空白，改行の追加などの変更は，構文に影響を与えることなく，主に見栄えを変更するために行われる．構文に影響を与えない変更としては，例えばプログラムを理解しやすくするためのコメントの追加やインデントの調整などが挙げられる (図 4.4).

識別子の宣言に影響を与えない構文の変更

文の削除や，すでに宣言されている識別子だけを利用した文の追加や式の変更は，識別子の宣言に影響を与えることなく，構文を変更する．このような変更として for 文を while 文に置き換えるような構文の変更 (図 4.5) や式の演算を変えるような変更が挙げられる.


```
switch (getchar()) {
  case 'a': ...
  case 'b': ...
  default: ...
}
```

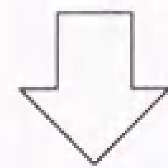


```
int c;

c = getchar();
fprintf(stderr, "%c\n", c);
switch (c) {
  case 'a': ...
  case 'b': ...
  default: ...
}
```

図 4.3: デバッグコードの追加

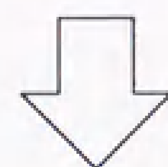
```
void func1() { char a; a = 'x';putchar(a); }
```



```
void func1()
{
    char a;
    a = 'x';
    putchar(a); /* output */
}
```

図 4.4: 構文に影響を与えない変更

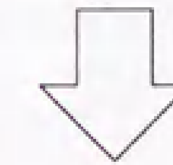
```
int i;
for (i = 0; i < 10; i++)
    printf("%d\n", i);
```



```
int i;
i = 0;
while (i < 10) {
    printf("%d\n", i);
    i++;
}
```

図 4.5: 識別子の宣言に影響を与えない構文の変更


```
void func3(int a)
{
    for (;;) {
        a--;
        if (a < 0) break;
    }
}
```



```
void func3(int a)
{
    int debug;
    debug = 0;
    for (;;) {
        debug++;
        fprintf(stderr, "%d\n", debug);
        a--;
        if (a < 0) break;
    }
}
```

図 4.6: 識別子の宣言に影響を与える変更

識別子の宣言に影響を与える変更

新たな変数を宣言したり，不要な識別子を削除したりといった変更は，識別子の宣言に影響を与える変更である．デバッグ用の変数とコードの追加 (図 4.6) や，その他多くの変更はほとんどが識別子の宣言に影響を与える変更である．

4.3 三層モデル

4.3.1 変更を行う際のユーザの視点

4.2 節において，ソースプログラムの変更は，

1. 構文に影響を与えない変更

2. 識別子の宣言に影響を与えない構文の変更

3. 識別子の宣言に影響を与える変更

の三つに分類できることを述べた．ソフトウェアの保守や部品化，あるいは再利用などにおいて，ユーザがソースプログラムを変更する場合，ユーザはこれらの三つの分類を意識してソースプログラムを変更する．すなわち，構文に影響を与えない変更を行う場合にはテキストレベルの視点において，識別子の宣言に影響を与えない構文の変更を行う場合には構文レベルの視点において，また識別子の宣言に影響を与える変更を行う場合にはスコープ規則に基づいて識別子を扱う視点においてソースプログラムを変更する．

テキストレベルの視点

ユーザが行う変更のうち，テキストレベルの視点において行われるものは，コメントの追加や削除，あるいはインデントの調整，空行の追加などである．これらの変更は，構文を意識せずに行うことができる．

構文レベルの視点

構文レベルの視点において行われるユーザの変更には，変数宣言の順序の変更や，文の順序の変更といったものが挙げられる．これらの変更は，構文規則に基づいて行われる．

スコープ規則に基づいて識別子を扱う視点

新たな変数などの追加や，あるいは既存の変数などの削除を行う場合は，ユーザは構文規則だけでなく，同時にスコープ規則を意識する必要がある．すなわ

ち，同名の変数が同一のスコープにおいて二重に定義されたり，あるいは未定義の変数が参照されたりといったことがないように変更しなければならない．

4.3.2 ソースプログラムの変更における制約

ソースプログラムの変更にともなう制約は，

1. 前処理や字句などの表現に関する制約

2. 構文規則に関する制約

3. スコープ規則や型の整合性に関する制約

の三つに分類することができる．この三つの制約は，ちょうど変更操作におけるユーザの三つの視点，

1. テキストレベルの視点

2. 構文レベルの視点

3. スコープ規則に基づいて識別子を扱う視点

にそれぞれ対応している．

ユーザの三つの視点や，三つの制約は，コンパイラの字句解析，構文解析，意味解析の各段階に一致している．そこで，一貫性を保ったソフトウェアの変更を実現するためのソフトウェアモデルとして，三層モデルを提案する．三層モデルは表現層 (Appearance Layer)，構文層 (Syntax Layer)，記号層 (Symbol Layer) の三層からなるソフトウェアモデルである．実際の解析の様子を図 4.7 に示す．

三層モデルでは，表現層において字句や空白あるいはコメントなどを，構文層において構文木を，記号層においてスコープ規則によって同一視された識別

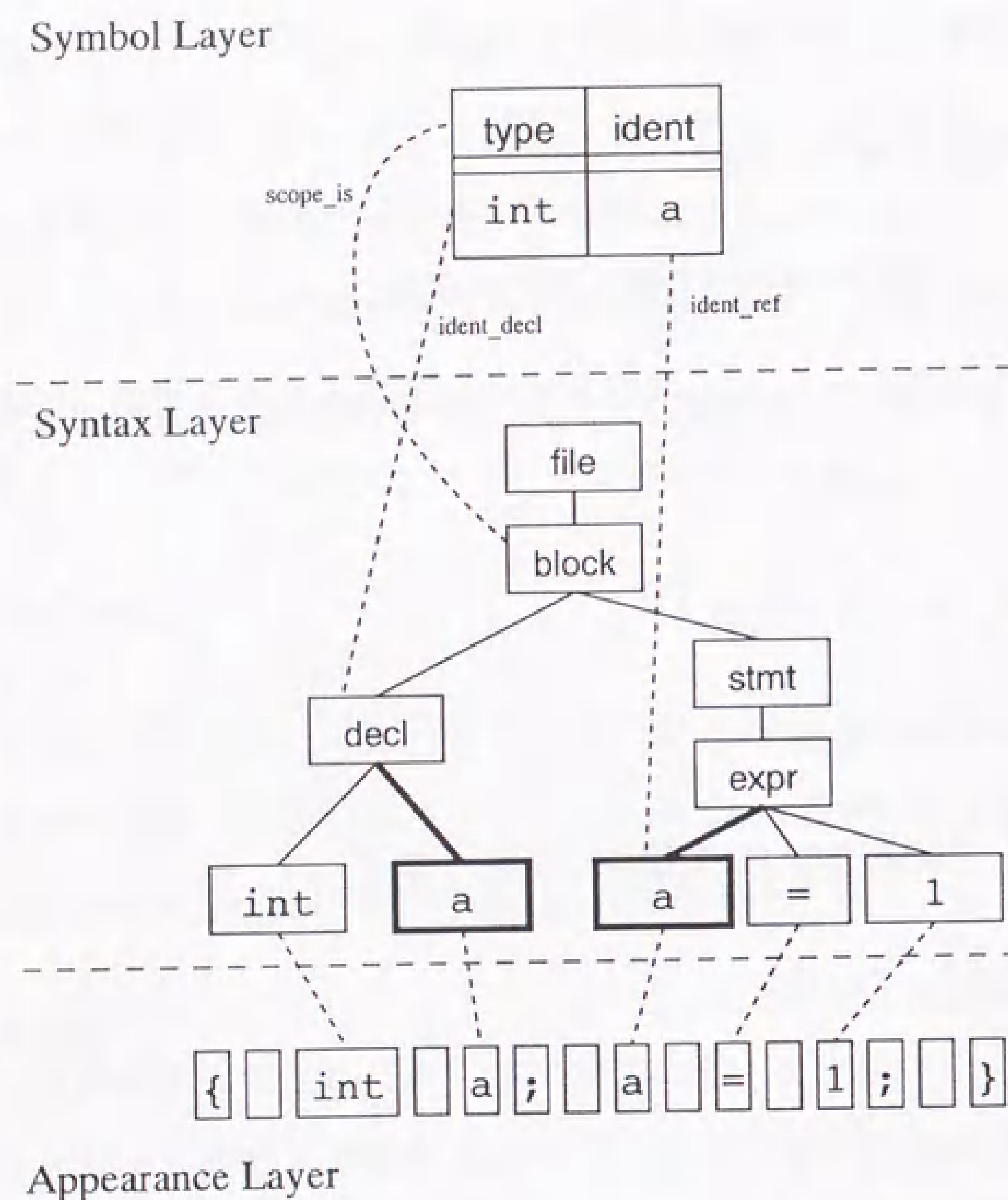


図 4.7: 三層モデルによる解析の例

子を管理し，ユーザの視点をリポジトリの構造によって明確に区別することにより，見通しの良い変更操作を実現可能にしている．

4.4 変更操作の例

ソフトウェアの生産性向上において，ソースプログラムを読みやすくすることは重要である．多くのプロジェクトにおいて，様々なプログラマが記述したプログラムのプログラミングスタイルを統一することは，プログラムを読みやすくするという点において有効である．

そこで，三層モデルによる変更操作の例として，様々なプログラマが記述したプログラムのスタイルを統一する場合を考える．

プログラミングスタイルの統一の例として，

- プログラミングスタイルにしたがったコメントの追加
- 変数宣言の方法の統一
- 変数名の変更

について，それぞれ必要な操作を考える．

また，より具体的な例として，

- 三層モデルを用いた既存のソフトウェアの再利用
- 三層モデルを用いたソフトウェアのデバッグ

について考察する．

4.4.1 コメントの追加

プログラミングスタイルにしたがったコメントの追加とは，例えば

- 変数の内容に関する説明は，その変数の宣言の直後にコメントを記述する

```
1: int x, y, z;
2: char a, b;
```

図 4.8: 変数宣言の例 1

```
1: int x;
2: int y;
3: int z;
4: char a;
5: char b;
```

図 4.9: 変数宣言の例 2

- 関数の内容に関する説明は，その関数本体の宣言の前の行にコメントを記述する

などである．コメントに対する変更は，プログラムの表現を変更するだけであり，プログラムの構文や意味に影響を与えない．三層モデルでは，これを表現層において行う．

4.4.2 変数宣言の方法の統一

図4.8と図4.9は同じ変数宣言であるが，図4.8は同じ用途をもつ変数をまとめて宣言することによってプログラムを読みやすくすることができ，また図4.9は各宣言を別々に分けることによってそれぞれにコメントをつけることができる．

図4.8のような宣言から図4.9のような宣言に変更する場合，構文は変更されるが，プログラムの意味に影響を与えない．このような変更は，三層モデルでは構文層において行う．


```
1: {
2:     int a, b;
3:     a = 1;
4:     {
5:         int a;
6:         a = 2;
7:     }
8:     b = a;
9: }
```

図 4.10: sample.c：2行目の変数 a を c に変更する

4.4.3 変数名の変更

変数名を変更する場合は，各変数の有効範囲を考慮する必要がある．三層モデルでは，この操作は記号層に対して行われ，記号層の変更が構文層および表現層に伝えられる．

図4.10において，2行目で宣言されている変数 a の名前だけを c に変更する場合を考える．この場合，6行目の変数 a は，スコープが異なるため，単純に文字列を置き換えることはできない．そこで，スコープ規則を考慮した変更が必要となる．

図4.10のプログラムを解析した場合，図4.11のようになる．ここで注目している2行目で宣言された変数 a は太線で示されているものである．そこで，この太線で示されている変数 a だけを c に変更することにより，5行目で宣言されている変数 a に影響を与えることなく，変数名を変更することができる．この変更は，構文層における対応する識別子を変更し，構文層の変更が表現層に反映される．

大域変数と局所変数や，あるいは構造体のメンバ名などの場合についても同様に扱うことができる．

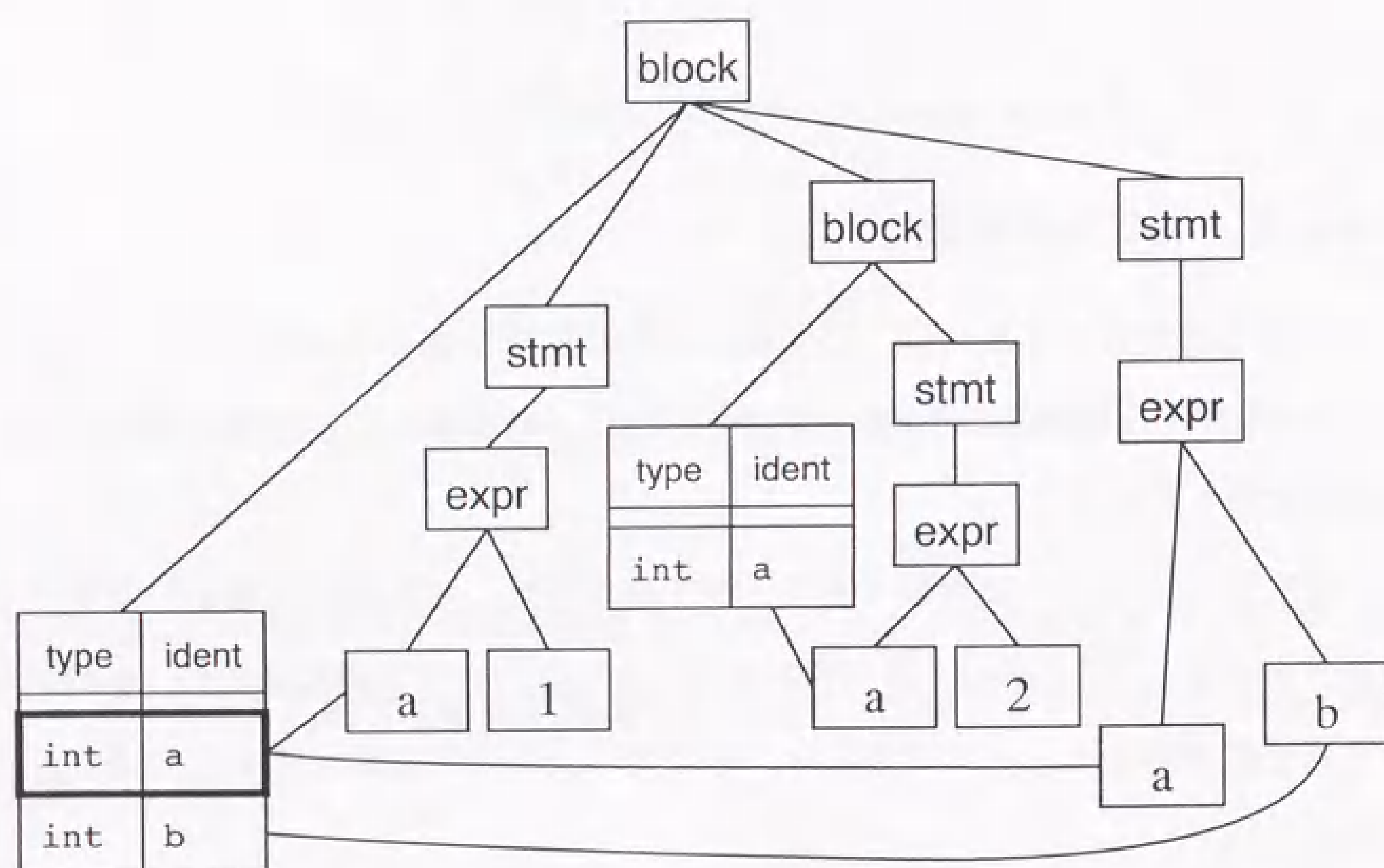


図 4.11: sample.c に対するインスタンス

4.4.4 ソースプログラムの再利用

既存のソースプログラムの再利用の例としてキューを管理するプログラムから，スタックを管理するプログラムを生成する場合を考える．

図4.12はキューを管理するプログラムの例である．変数 num は要素の数を表し，変数 q_in および q_out はキューへの入力位置およびキューからの出力位置を表す．スタックの場合には，入力位置および出力位置は要素の数と同じであり，図4.12のプログラムにおいて22行目の “q_in” および35行目の “q_out” を要素の数を表す “num” に変更すればよい．22行目の文 “queue[q_in] = x;” に対する記号層のインスタンスは図4.13のようになる．

そこで，22行目および35行目の変数参照を図4.13の点線のように変更することにより各式から参照される変数を num に変更する．

次に，変数 queue の名前を stack に，また関数 init_queue()，enqueue()，dequeue() の名前をそれぞれ init_stack()，push()，pop() に変更する．このような変更は，4.2節における変更の三つの分類のうち，識別子の宣言に影響を与える変更にあたり，三層モデルでは記号層において行う．

ここで，変数 stack に関して依存解析を行うと，変数 q_in および q_out が不要であることがわかる．そこで，変数 q_in および q_out に関わる文をすべて削除する．図4.12のプログラムでは，11, 12, 20, 21, 22, 33, 34, 35 行目の各文がこれに対応する．このような変更は，4.2節における変更の三つの分類のうち，識別子の宣言に影響を与えない構文の変更にあたり，三層モデルでは構文層において行う．また，この削除により変数 q_in および q_out の宣言が不要となり，これらの宣言を削除する．図4.12のプログラムでは，5行目と6行目の宣言がこれに対応する．このような変更は識別子の宣言に影響を与える変更であり，記号層において行う．

最後にインデントを調整すると，キューを管理するプログラムから，図4.14の

```c
 1 #define SIZE    100
 2 
 3 static int queue[SIZE];
 4 static int num;
 5 static int q_in;
 6 static int q_out;
 7 
 8 void init_queue()
 9 {
10     num = 0;
11     q_in = 0;
12     q_out = 0;
13 }
14 
15 void enqueue(int x)
16 {
17     if (num >= SIZE) {
18         error();
19     } else {
20         q_in++;
21         if (q_in == SIZE)
22             q_in = 0;
23         queue[q_in] = x;
24         num++;
25     }
26 }
27 
28 int dequeue()
29 {
30     if (num <= 0) {
31         error();
32     } else {
33         q_out++;
34         if (q_out == SIZE)
35             q_out = 0;
36         num--;
37         return queue[q_out];
38     }
39 }
```

図 4.12: queue.c

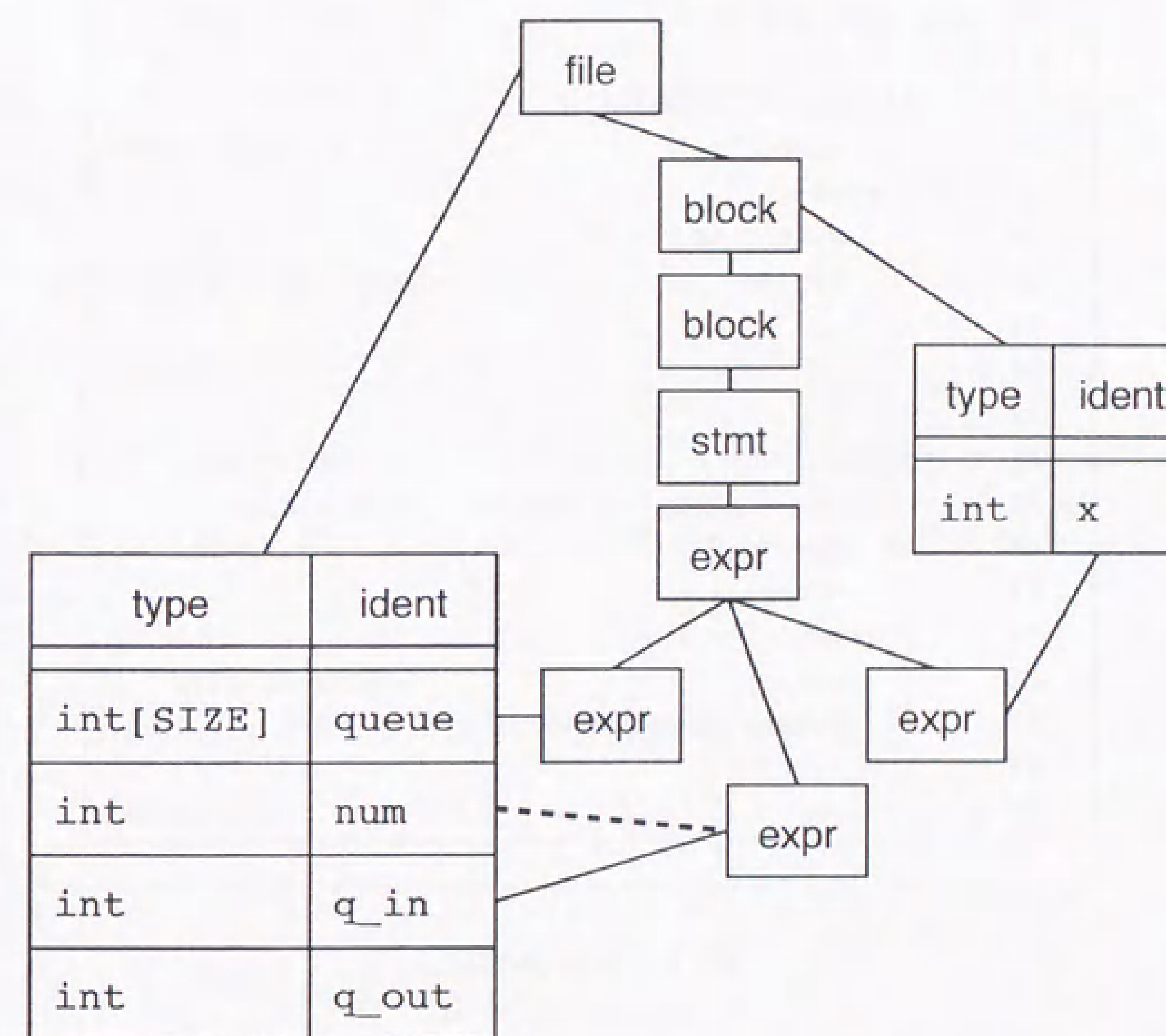


図 4.13: queue[q_in] = x; に対するインスタンス

```
 1 #define SIZE    100
 2
 3 static int stack[SIZE];
 4 static int num;
 5
 6 void init_stack()
 7 {
 8     num = 0;
 9 }
10
11 void push(int x)
12 {
13     if (num >= SIZE) {
14         error();
15     } else {
16         stack[num] = x;
17         num++;
18     }
19 }
20
21 int pop()
22 {
23     if (num <= 0) {
24         error();
25     } else {
26         num--;
27         return stack[num];
28     }
29 }
```

図 4.14: stack.c

ようなスタックを管理するプログラムを得ることができる．インデントの調整は，構文に影響を与えない変更であり，三層モデルでは表現層において行う．

4.4.5 デバッグコードの挿入

プログラムのデバッグを行う際に，その一つの方法として変数の値の変化を追いかけることが行われる．

例えば，ある変数の値を出力するための関数を用意しておき，その変数の値が変更されるごとに，用意した関数を呼び出すコードを追加するという方法で


```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int fib(int x)
 5 {
 6     int fib1 = 0, fib2 = 1, tmp;
 7
 8     while (fib1 <= x) {
 9         tmp = fib2;
10         fib2 = fib1 + fib2;
11         fib1 = tmp;
12     }
13
14     return (fib1);
15 }
16
17 int main(int argc, char *argv[])
18 {
19     int x;
20
21     if (argc < 2) {
22         fprintf(stderr, "usage: %s num\n", argv[0]);
23         exit(1);
24     }
25
26     x = atoi(argv[1]);
27     printf("x = %d, fib = %d\n", x, fib(x));
28
29     return (0);
30 }
```

図 4.15: fibonacci.c

ある．このような変更の場合，小さなプログラムでは手で行ってもそれほど問題にはならないが，プログラムが大きくなるにつれて手作業では困難になり，また新たなバグを混入する要因ともなる．そこで，このようなデバッグコードの挿入を一貫性を保ちながら自動的に行うことが望まれる．

入力された値を越えない最大のフィボナッチ数を求めるプログラムとして，図 4.15 のプログラムが与えられた場合を考える．このプログラムに入力として “4” を与えて動作させる (図 4.16) と出力は “5” となり，「入力された値を越え

```
% fib 4
x = 4, fib = 5
```

図 4.16: 入力に 4 を与えて実行した結果

ない」という条件にあっていない．そこで，このプログラム “fibonacci.c” を変数の値を追いかけることにより，デバッグすることを考える．

図4.15のプログラムにおいて，関数 fib() の返り値は，14 行目の変数 fib1 である．そこで，変数 fib1 の値に注目して，変数 fib1 への値の代入のすべてにおいて依存解析を行い，変数 fib1 および変数 fib1 に影響を与えるすべての変数の値を，代入された直後に出力する．

このような変数の値を出力するためのコードの追加は，識別子の宣言に影響を与えない構文の変更であり，三層モデルでは出力するための文を構文層において追加する．図4.17 のプログラムは，文を追加した後，表現層においてインデントを調整したものである． このプログラムに入力として “4” を与えて動作させると図4.18のようになる．この結果から関数 fib() の while 文の条件式において変数 x は変数 fib1 ではなく，変数 fib2 と比較しなければならないと推測できる．そこで，記号層において while 文の条件式の変数 fib1 への参照を変数 fib2 への参照に変更する．

次に，変更されたプログラムに入力として “5” を与えて動作させると図4.19のように出力は “5” となり，「入力された値を越えない」という条件にあわない．この結果から関数 fib() の while 文の条件式において変数 x と変数 fib2 とは “<=” による比較ではなく，“<” によって比較されなければならないと推測できる．そこで，構文層において while 文の条件式の演算を “<=” による比較から “<” による比較に変更する．


```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int fib(int x)
 5 {
 6     int fib1 = 0, fib2 = 1, tmp;
 7     fprintf(stderr, "fib1 = %d\n", fib1);
 8
 9     while (fib1 <= x) {
10         tmp = fib2;
11         fib2 = fib1 + fib2;
12         fib1 = tmp;
13         fprintf(stderr,
14                 "fib1 = %d, fib2 = %d, tmp = %d, x = %d\n",
15                 fib1, fib2, tmp, x);
16     }
17
18     return (fib1);
19 }
20
21 int main(int argc, char *argv[])
22 {
23     int x;
24
25     if (argc < 2) {
26         fprintf(stderr, "usage: %s num\n", argv[0]);
27         exit(1);
28     }
29
30     x = atoi(argv[1]);
31     printf("x = %d, fib = %d\n", x, fib(x));
32
33     return (0);
34 }
```

図 4.17: デバッグコードを追加した fibonacci.c

```
% fib 4
fib1 = 0
fib1 = 1, fib2 = 1, tmp = 1, x = 4
fib1 = 1, fib2 = 2, tmp = 1, x = 4
fib1 = 2, fib2 = 3, tmp = 2, x = 4
fib1 = 3, fib2 = 5, tmp = 3, x = 4
fib1 = 5, fib2 = 8, tmp = 5, x = 4
x = 4, fib = 5
```

図 4.18: デバッグコードを追加した fibonacci.c の実行結果

```
% fib 5
fib1 = 0
fib1 = 1, fib2 = 1, tmp = 1, x = 5
fib1 = 1, fib2 = 2, tmp = 1, x = 5
fib1 = 2, fib2 = 3, tmp = 2, x = 5
fib1 = 3, fib2 = 5, tmp = 3, x = 5
fib1 = 5, fib2 = 8, tmp = 5, x = 5
x = 5, fib = 5
```

図 4.19: 入力として 5 を与えた場合の実行結果

このようにして，三層モデルを利用して，「入力された値を越えない最大のフィボナッチ数を求める」プログラムをデバッグすることができる．

4.5 まとめ

ソフトウェアの保守や部品化，再利用において，ソフトウェアを変更する場合，変更後も，構文規則や意味規則に関して一貫性が保たれている必要がある．ソフトウェアは一般にソースプログラムだけではなく，各種仕様書やマニュアルなどの様々なドキュメントから構成される．しかし，ソースプログラム以外のドキュメントは各開発現場ごとに書式が異なるなどこれらを統一的に扱うことは困難である．そこでソースプログラムの変更に的を絞り，ユーザの視点および変更にともなう制約からソースプログラムの変更を考察し，それぞれが三つに分類できることを述べた．

また，ユーザの視点および変更にともなう制約から，ソフトウェアを一貫性を保ちながら変更するためのソフトウェアモデルとして，三層モデルを提案した．三層モデルは，ソフトウェアを表現層，構文層，記号層の三層によって表現するソフトウェアモデルである．この三層モデルにおいて，各層で三つの制約を別々に扱い，またユーザが変更する際の視点を各層に分けることにより，一貫性を保ったソースプログラムの変更操作を見通し良く行うことができる．このため，リポジトリを用いたソフトウェアの保守や部品化，再利用などに有効である．

第 5 章

再利用によるソフトウェア開発

5.1　はじめに

近年，多くのコンピュータシステムが開発され，その規模は大きくなっている．コンピュータシステムは，システムが稼働し始めてからも，そのシステムを取り巻く環境は日々変化する．したがって，環境の変化に対応するために，システムを修正し新たな機能を追加することが必要となる．

ある既存のシステムに対して，環境の変化などから，新システムへの変更の必要が生じた場合，新しいシステムを作成する方法として，

- 新たに一から作成する方法
- 既存システムに変更を加えることによって新システムへ進化させる方法

の二つの方法が考えられる．一般に，上述のような場合には，

- システムの大部分は同じ機能を提供する
- 過去の過ちを繰り返さないことが重要となる

などから，後者の方法が望ましい場合が多い．

本章では，既存システムに変更を加えることによって新システムへ進化させる方法に着目する．この際，既存システムのソースプログラムに直接変更を加えるのではなく，仕様の段階において変更を十分に吟味してから，それに基づいてソースプログラムを変更することが重要である．

一方，既存システムと作成すべき新しいシステムとの間の差分を表現する手法としてシナリオに基づいた手法 [11] が提案されている．また仕様からソースプログラムを生成する研究として，UML のステートチャートに注目した研究 [27] や，同じく UML のコラボレーション図に注目した研究 [5] などが行われている．

本章では，再利用によってソフトウェアを開発するための枠組みを提案する．また，この枠組みに基づいて，第 3 章で述べた Sapid を用いて必要な情報を取得することにより，既存システムから新システムを構築する例を挙げる．

5.2　再利用によるソフトウェア開発の枠組み

既存ソフトウェアの再利用においては，ソフトウェアを変更する技術とともに，既存のソフトウェアから再利用可能な部分を見つける方法が問題となる．既に，既存システムと作成すべき新しいシステムの間の差分を表現するのに適した手法として，シナリオに基づいた手法 [11] が提案されている．

一方，UML [6, 32] は，オブジェクト指向で業務を分析したり，システムを開発したり，ソフトウェアモジュールを設計したりする際の図の描き方・記法を定めている [9]．オブジェクト指向によるシステムの開発においては，システムを様々な側面から記述し分析する必要があり，UML では様々な図を定義している．それらは，システムの静的な構造に焦点を当てた構造図，システムの動的な変化や相互作用に焦点を当てた振舞い図，実装時の構成や実行時の配置に焦点を当てた実装図の 3 つに分類できる．UML を用いたシステムの開発は，ユースケースやシナリオを中心に行われる．ユースケースはシステムの利用の仕方をシステム外部の視点から記述したものであり，シナリオはユースケースの具体的な事例である．ユースケースやシナリオから，オブジェクトを抽出し，1 つ

のオブジェクトの振舞いや，複数のオブジェクトの相互作用などを分析し，設計を行う．

あるシステムが既に稼働しており，環境の変化などからシステムを変更する必要の生じた場合，新システムを作成する方法としては，新たに一から作成する方法と，既存システムに変更を加えることによって進化させる方法の，二つの方法が考えられる．ここでは後者の方法に注目し，ユースケースやシナリオを中心に，既存のシステムを再利用し，進化させることによって新しいシステムを開発するための枠組みを提案する．ここで想定している状況を次に示す．

- あるシステムが存在し，既に稼働している．
- 環境の変化やシステムの不備などから，そのシステムに不満があり，システムを変更したいと考えている．
- システムの実行ファイルだけでなく，そのソースプログラムやシステムの使用法などのドキュメントがそろっている．

すなわち，あるユーザが利用しているシステムに不満がある．そのユーザはシステムのソースプログラムに手を入れることができ，実際の実行やドキュメントなどを通じて使用方法を知ることができる．

図5.1は，提案する再利用によるソフトウェア開発の枠組みを示している．この枠組みは主に次の三つのステップからなる．

1. 既存システムからの要求の抽出
2. 新システムの仕様の記述
3. 新システムの実現

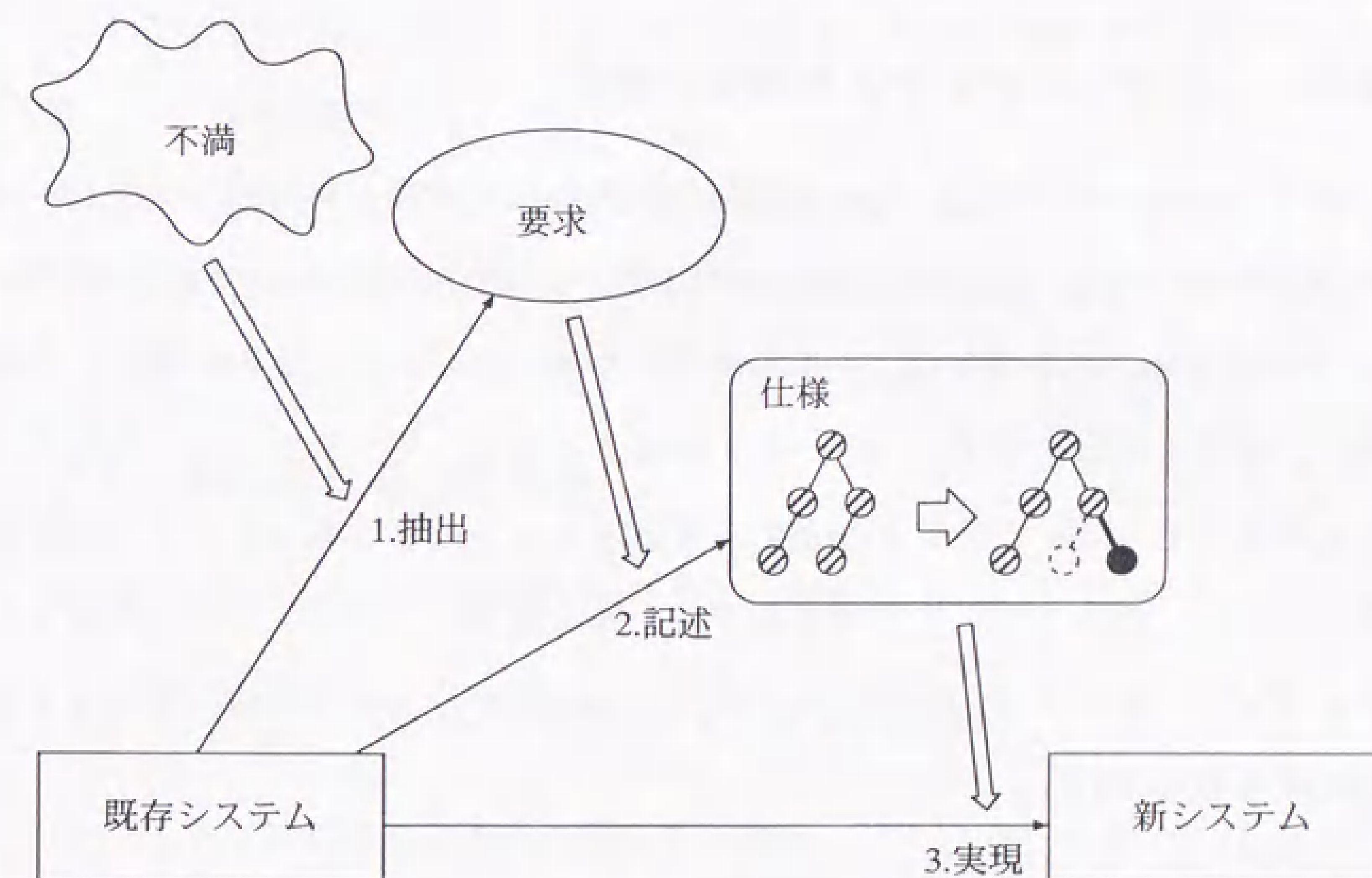


図 5.1: 再利用によるソフトウェア開発の枠組み

まず，システムに対する不満を基に既存システムを解析することによって，新システムに対する要求を抽出する．次に，新システムに対する要求を基に既存システムを解析して，新システムの仕様を既存システムの仕様からの差分という形で記述する．最後に，仕様の差分を基に既存システムに変更を加えるなどして新システムを実現する．仕様の記述にはUMLを用い，シナリオを中心に分析することによって再利用の可能な部分，および変更の必要な部分を見つける．以下の節では，各ステップについて詳しく説明する．

5.2.1　既存システムからの要求の抽出

まず，あるユーザが抱いている既存システムへの不満から，新システムへの要求を明確にする．そこで既存システムのシナリオを記述する．ユーザは既存システムに対する不満から，その不満のある部分について，実際に実行してみたり，使用方法などのドキュメントを参照するなどして，シナリオを記述することができる．このシナリオを基に，新システムではどうなっているべきかを考えながら，新システムに対するシナリオを記述する．既存システムに対するシナリオと，新システムに対するシナリオとの差が，ユーザが抱いている不満に対する要求である．

5.2.2　新システムの仕様の記述

要求を基に，既存システムを解析することによってシステムの仕様を記述する．仕様の記述にはUMLで定義されている各種の図を用いる．UMLでは，ソフトウェアシステムの静的な構造や，振舞いを記述するために様々な図を用意している．新システムへの要求を中心に，既存システムの仕様を記述することによって，既存システムの問題点を明確にすることができる．ここで重要なのは，既存システムに対して記述した仕様の各部分と，それに対応するソースプログラムの各部分が関係付けられていることである．その一つの方法としては，

関数名や変数名などの名前による関連付けが考えられる．

さらに，記述した既存システムに対するUMLの各図を修正することによって，新システムの仕様を記述する．この際，各図の修正は，要素の追加や削除という形で行う．これらの仕様から，既存システムに対する変更点を得ることができる．

5.2.3　新システムの実現

仕様の差分を基に，既存システムに適宜追加・修正・削除を行うことによって新システムを実現する．

先に述べたように，既存システムに対する仕様と既存システムのソースプログラムは名前などによって関連付けられている．この関連を頼りに，既存システムと新システムの仕様の差分をソースプログラムに反映することができる．

5.2.4　枠組みに必要な機能

提案した枠組みに必要な機能は以下のようである．

- 既存システムのソースプログラムを解析する機能
- ソースプログラムの仕様を構成する機能
- 再利用可能な部品を抽出する機能

既存システムを再利用するためには，既存システムのソースプログラムを解析することが必要である．その際，字句解析や構文解析だけでなく，データの流れや制御の流れといった依存解析なども重要となる．また，解析の結果からソースプログラムの仕様を構成する機能や，既存のソースプログラムから，再利用可能と判断された部品を一貫性を保ちながら抽出する機能なども重要である．

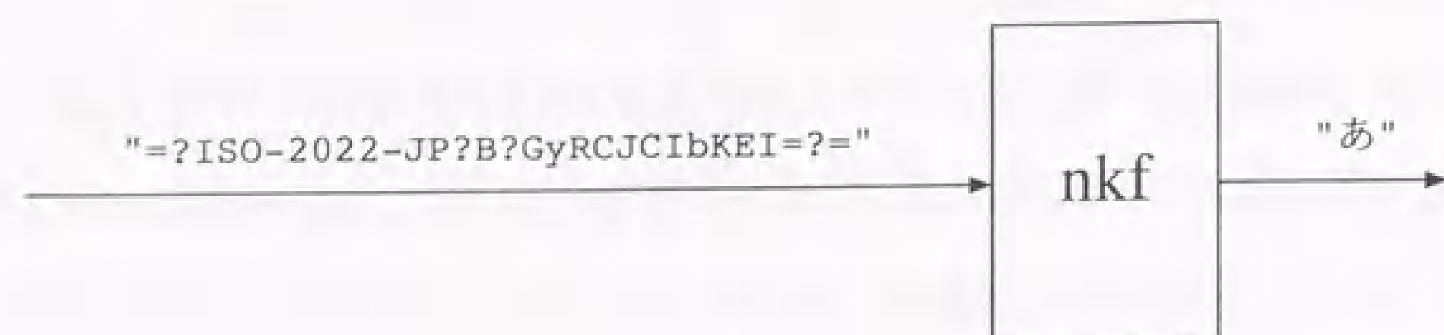


図 5.2: nkf による復号化

以降の節では，第3章で述べた CASE ツール・プラットフォーム Sapid と提案した枠組みの関係について具体的な例を用いて考察する．

5.3 Sapid に基づいたソフトウェア再利用

Sapid に基づいたソフトウェア再利用の例を示す．ここでは，対象ソフトウェアとして，漢字コード変換フィルタ nkf を考える．nkf は，計算機上において，JIS，SJIS，EUC といった様々な形式で表現される漢字コードを相互に変換する機能を始め，多くの機能を持っているが，その一つに "=?ISO-2022-JP?B?" で始まるストリームを，MIME base64 で符号化されたストリームであるとみなして，これを復号化して出力する機能がある．

例えば，図 5.2 に示すように，nkf に入力ストリームとして，

"=?ISO-2022-JP?B?GyRCJCIbKEI=?="

を与えると，"=?ISO-2022-JP?B?" に続くストリームを base64 で復号し，JIS コードの "あ" として出力する．

base64 による符号化は，データを 24 ビットごとに分割し，これを 4 つの符号化された文字からなる文字列に変換することによって行われる．処理は左から右に 24 ビットのデータを 6 ビットごとの 4 つのグループに分割し，各 6 ビットのグループは対応する 64 文字のいずれかの文字に置き換えられる．base64 で

表 5.1: base64 で用いられる符号とその値の対応

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	'A'	17	'R'	34	'i'	51	'z'
1	'B'	18	'S'	35	'j'	52	'0'
2	'C'	19	'T'	36	'k'	53	'1'
3	'D'	20	'U'	37	'l'	54	'2'
4	'E'	21	'V'	38	'm'	55	'3'
5	'F'	22	'W'	39	'n'	56	'4'
6	'G'	23	'X'	40	'o'	57	'5'
7	'H'	24	'Y'	41	'p'	58	'6'
8	'I'	25	'Z'	42	'q'	59	'7'
9	'J'	26	'a'	43	'r'	60	'8'
10	'K'	27	'b'	44	's'	61	'9'
11	'L'	28	'c'	45	't'	62	'+'
12	'M'	29	'd'	46	'u'	63	'/'
13	'N'	30	'e'	47	'v'		
14	'O'	31	'f'	48	'w'	(pad)	'='
15	'P'	32	'g'	49	'x'		
16	'Q'	33	'h'	50	'y'		

用いられる符号とその値の対応を表 5.1 に示す．例えば "0x1B,0x24,0x42" という 24 ビットのデータを符号化すると，図 5.3 に示すように左から 6 ビットごとに分割され，"GyRC" という文字列になる．24 ビットごとに分割した場合，最後のグループが 24 ビットに足りない場合には符号化の際に '=' が追加される．したがって，最後のグループが 16 ビットの場合には，3 つの符号と 1 つの '=' によって，また最後のグループが 8 ビットの場合には，2 つの符号と 2 つの '=' によって符号化される．"0x28,0x42" という 16 ビットのデータを符号化すると，図 5.4 に示すように，"KEI=" という文字列になる．

nkf はバージョン 1.5 において，入力された "=?ISO-2022-JP?B?" に続くストリームが，base64 で符号化された正しいコードで構成されているかどうかに関

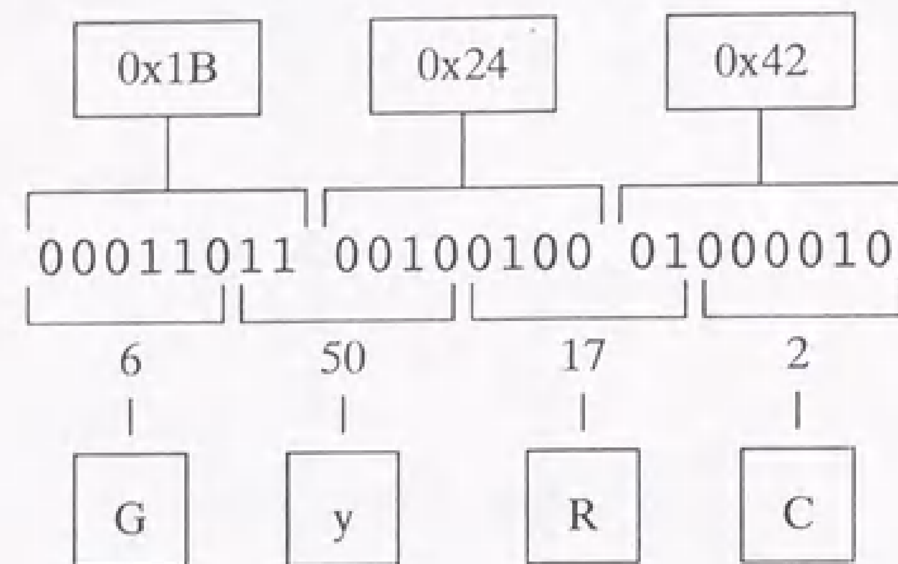


図 5.3: base64 による符号化の例 1

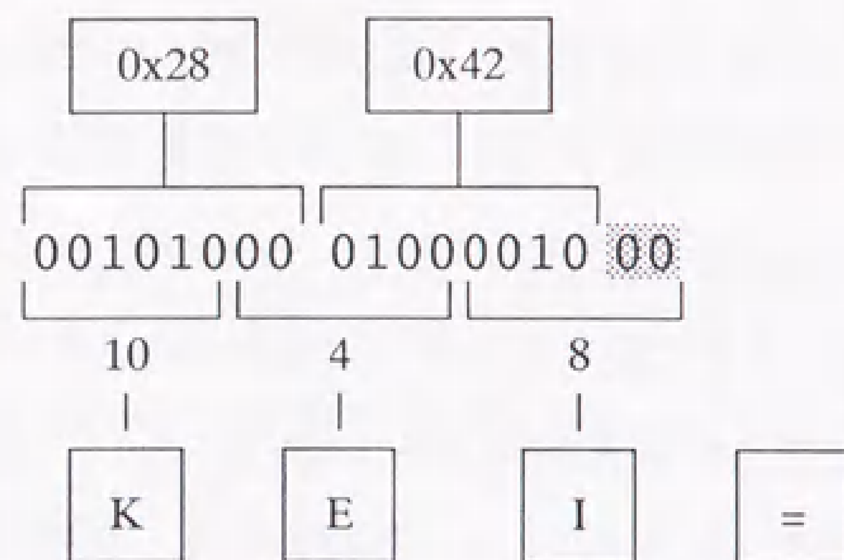


図 5.4: base64 による符号化の例 2

わらず，base64 で復号化し，結果を出力していた．

その後のバージョンでは，“=?ISO-2022-JP?B?” に続く入力ストリームが，base64 で符号化された正しいコードで構成されているものかどうかを検査する機能が追加されている．

そこで，nkf1.5 に対して，入力に “=?ISO-2022-JP?B?” を含むストリームが入力された場合に，続くストリームが base64 で符号化された正しい入力かどうかを検査する機能を追加した new_nkf を作成する場合を考える．

nkf1.5 は，C 言語を用いて約 1,500 行で記述されている．

5.3.1 シナリオの記述

まず初めに，nkf に様々な入力を与えることによってシステムがどのように動作するかをシナリオとして記述する．ここでは，“=?ISO-2022-JP?B?” から始まり，表 5.1 で示されている符号によって構成されたストリームが入力された場合と，表 5.1 以外の文字を含むストリームが入力された場合を考える．

シナリオ 1-1 nkf に入力ストリームとして正しいコードで構成されている “=?ISO-2022-JP?B?GyRCJCIbKEI=?=” を与えると，base64 で復号化して JIS コードで “あ” を出力する．

シナリオ 1-2 nkf に入力ストリームとして不正なコード ‘*’ が含まれている “=?ISO-2022-JP?B?GyRC*CIbKEI=?=” を与えると，base64 で復号化して不正な出力をする．

これらのシナリオは実際に nkf を実行するなどして記述することができる．

これに対して，新システムで要求されるシナリオを記述すると，

シナリオ 2-1 nkf に入力ストリームとして正しいコードで構成されている “=?ISO-2022-JP?B?GyRCJCIbKEI=?=” を与えると，“=?ISO-2022-JP?B?”

に続くストリームが正しいコードで構成されているかどうかを検査し，正しいコードで構成されているので，base64 で復号化して JIS コードで “あ” を出力する．

シナリオ 2-2 nkf に入力ストリームとして不正なコード ‘*’ が含まれている “=?ISO-2022-JP?B?GyRC*CIbKEI=?=” を与えると，“=?ISO-2022-JP?B?” に続くストリームが正しいコードで構成されているかどうかを検査し，不正なコード ‘*’ が含まれているので，復号化せずに入力されたストリームをそのまま出力する．

のようになる．

既存システムに対するシナリオ 1-1，シナリオ 1-2 と，新システムに対するシナリオ 2-1，シナリオ 2-2 との差分は，入力ストリームをそのまま復号化するのではなく，“=?ISO-2022-JP?B?” に続くストリームが正しいコードで構成されているかどうかを検査し，正しいコードで構成されていれば復号化して出力し，不正なコードが含まれていれば入力されたストリームをそのまま出力することである．

5.3.2 仕様の記述

5.3.1 節のシナリオの差分を基に，今回の変更で注目している部分を中心に既存システムの仕様を UML の各図を用いて記述する．既存システムのうち，ここで記述されないような部分については，既存システムから新システムへの変更において影響のない部分である．

要求を中心に，既存システムの仕様を記述するためには，シナリオの差分から，既存システムのソースプログラムの着目すべき部分を見つけ，関数間の呼び出しの関係や，大域変数の定義・利用の関係などを解析することが必要である．Sapid は，字句・構文解析の結果や依存解析の結果を参照する機能を提供

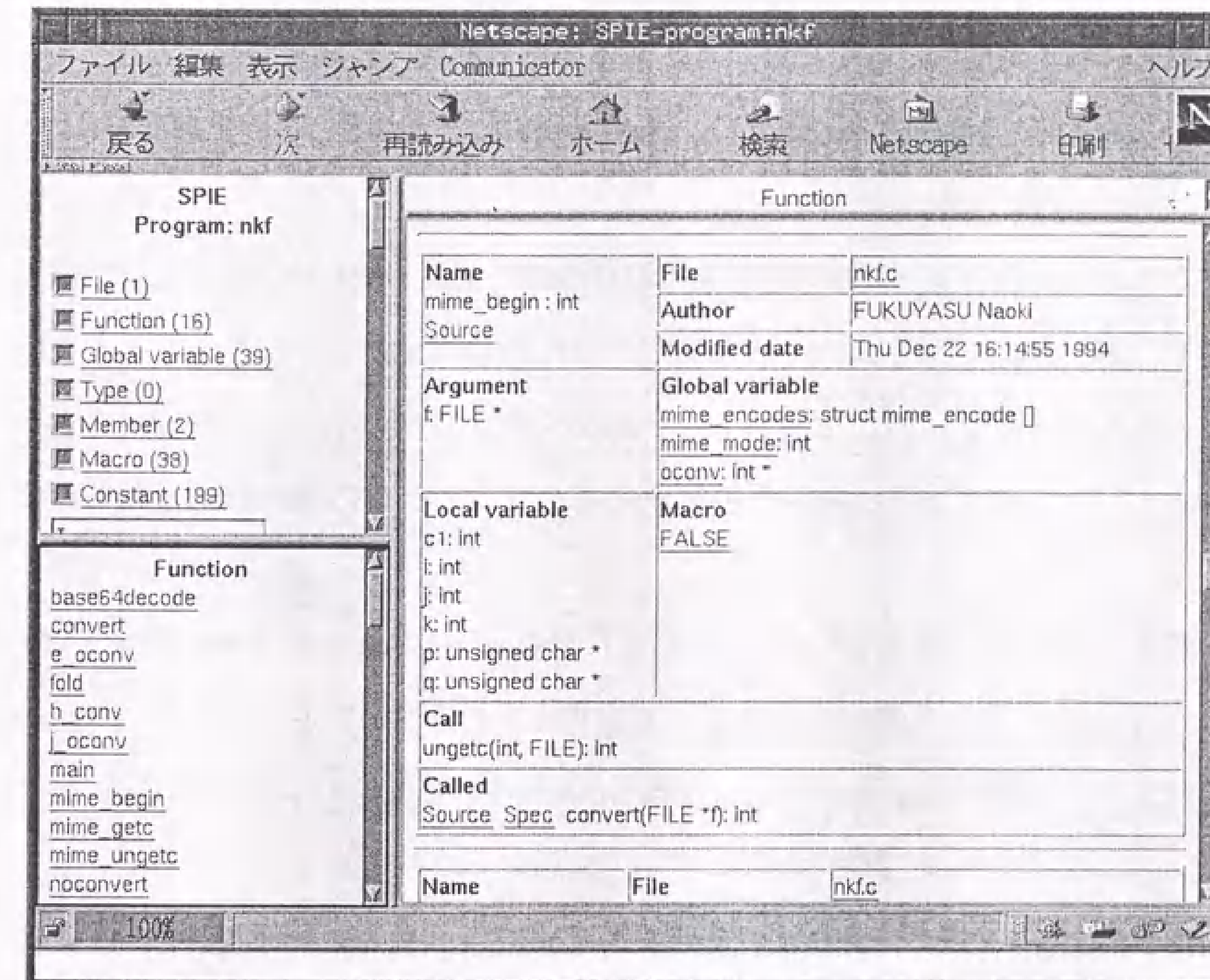


図 5.5: SPIE による関数 mime_begin() の出力

しており，これらの解析情報を簡単に取得することができる．

nkf1.5 のソースから，Sapid を利用してストリーム “=?ISO-2022-JP?B?” を処理している部分を見つけると，大域変数 mime_encodes が，このストリームを扱っていることがわかる．また，大域変数 mime_encodes を利用しているのは関数 mime_begin() である．

Sapid を利用した既存システムのソースプログラムを理解するためのツールである SPIE を用いると，関数 mime_begin() は，図 5.5 のように出力される．

ここから，関数 mime_begin() を呼び出しているのは，関数 convert() であ

ることなどがわかる．また，SPIE を用いて，関数 mime_begin() を基に，利用されている大域変数，さらにそれらに関連のある関数・変数を挙げると，mime_getc()，base64decode()，mime_mode などとなる．SPIE の詳細については，5.4.1 節で述べる．

これらの関数および変数の関係を UML の図で記述したものを図 5.6，図 5.7 に示す．図 5.6 は MIME のモードを判定するための部分をステートチャートで記述したものである．また，図 5.7 は "=?ISO-2022-JP?B?GyRCJCIbKEI=?=" というストリームが入力された場合の処理の流れをシーケンス図で記述したものである．

関数 convert() は入力 "=?" に対して，関数 mime_begin() を呼び出す．関数 mime_begin() は，以降の入力によって符号化のモードを判定し，変数 mime_mode に設定する．関数 convert() は，変数 mime_mode に設定されたモードにしたがって，入力ストリームを復号化する．

さらに，5.3.1 節のシナリオの差分を基に nkf1.5 の仕様に修正を加えることにより，new_nkf の仕様を記述する．

今回の変更では，入力ストリームが "=?ISO-2022-JP?B?" で始まる場合に，以降のストリームが base64 による正しい符号で構成されているかどうかを検査するための機能を追加する．このような機能を追加した図を図 5.8 および図 5.9 に示す．これらの図は図 5.6 および図 5.7 に変更を加えたものである．

図中の太線で示されている部分は新システムの仕様に新たに追加された部分であり，破線で示されている部分は新システムの仕様において削除された部分である．

nkf1.5 では，関数 mime_begin() によって MIME のモードが判定された後，以降の入力を "?=" が現れるまで順に復号化して出力していた．new_nkf では，入力ストリームが正しいコードで構成されているかどうかを検査するために，

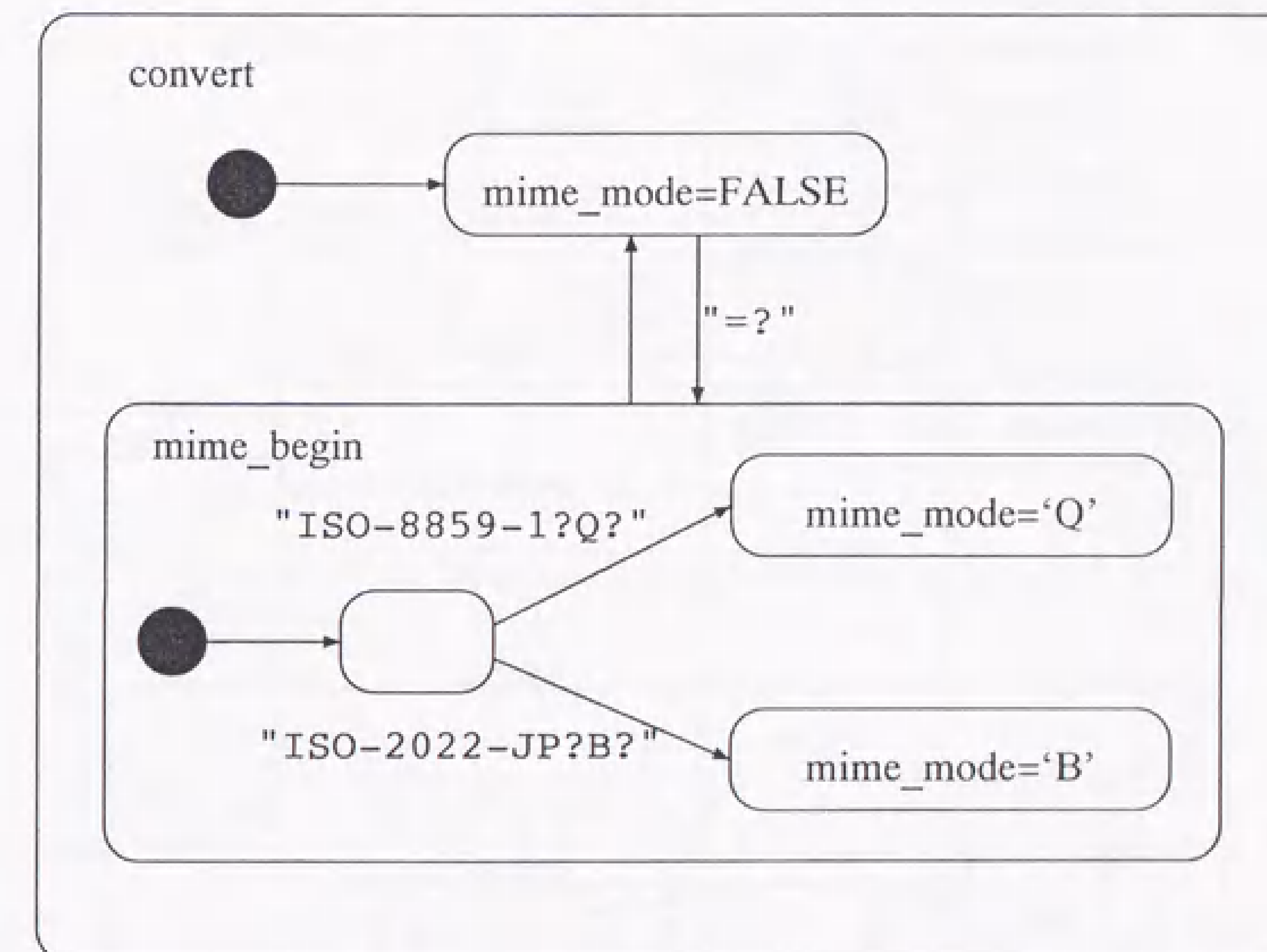


図 5.6: 既存システムのステートチャート

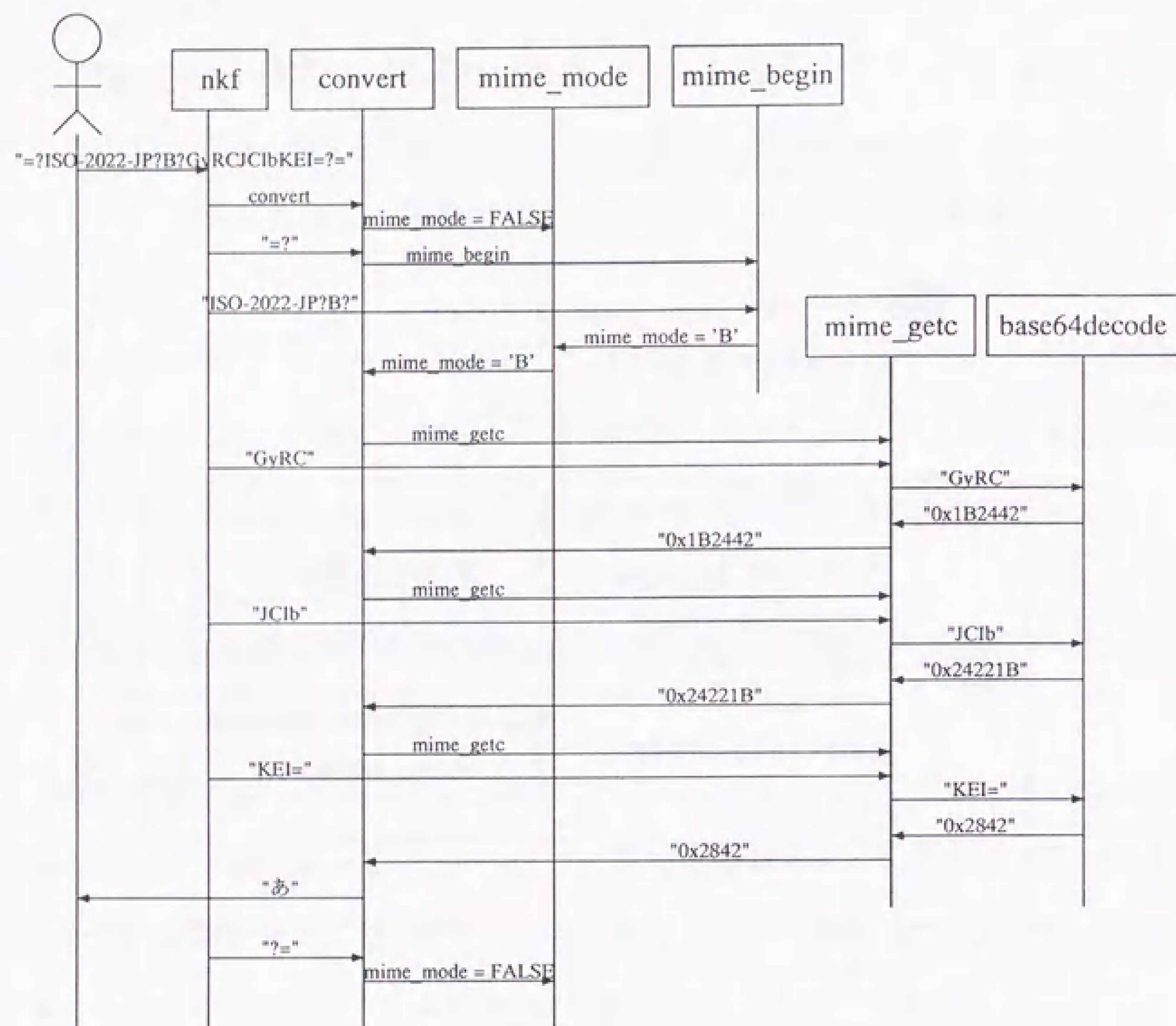


図 5.7: 既存システムの入力 "=?ISO-2022-JP?B?GyRCJCIbKEI=?=" に対するシーケンス図

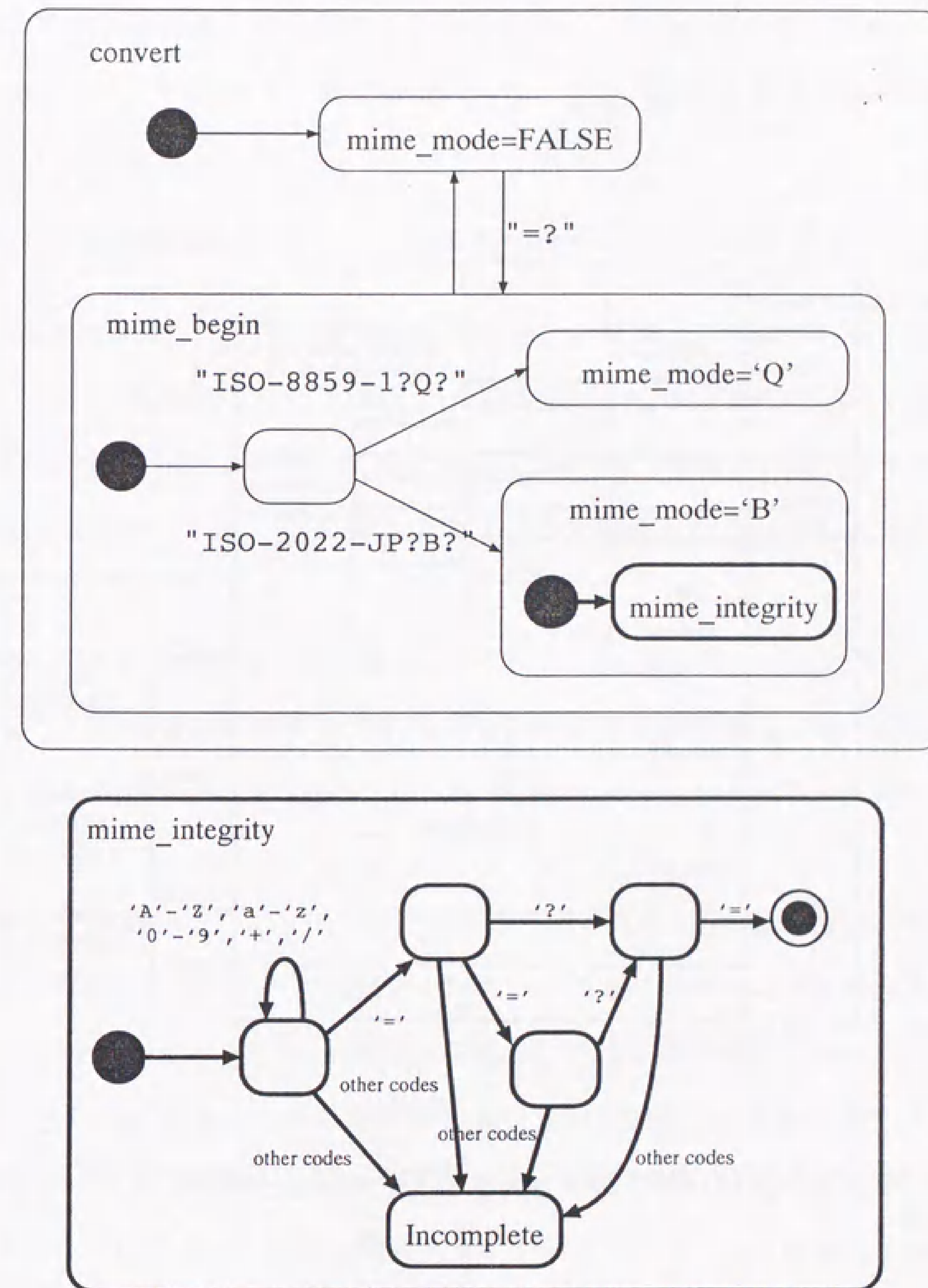


図 5.8: 新システムのステートチャート

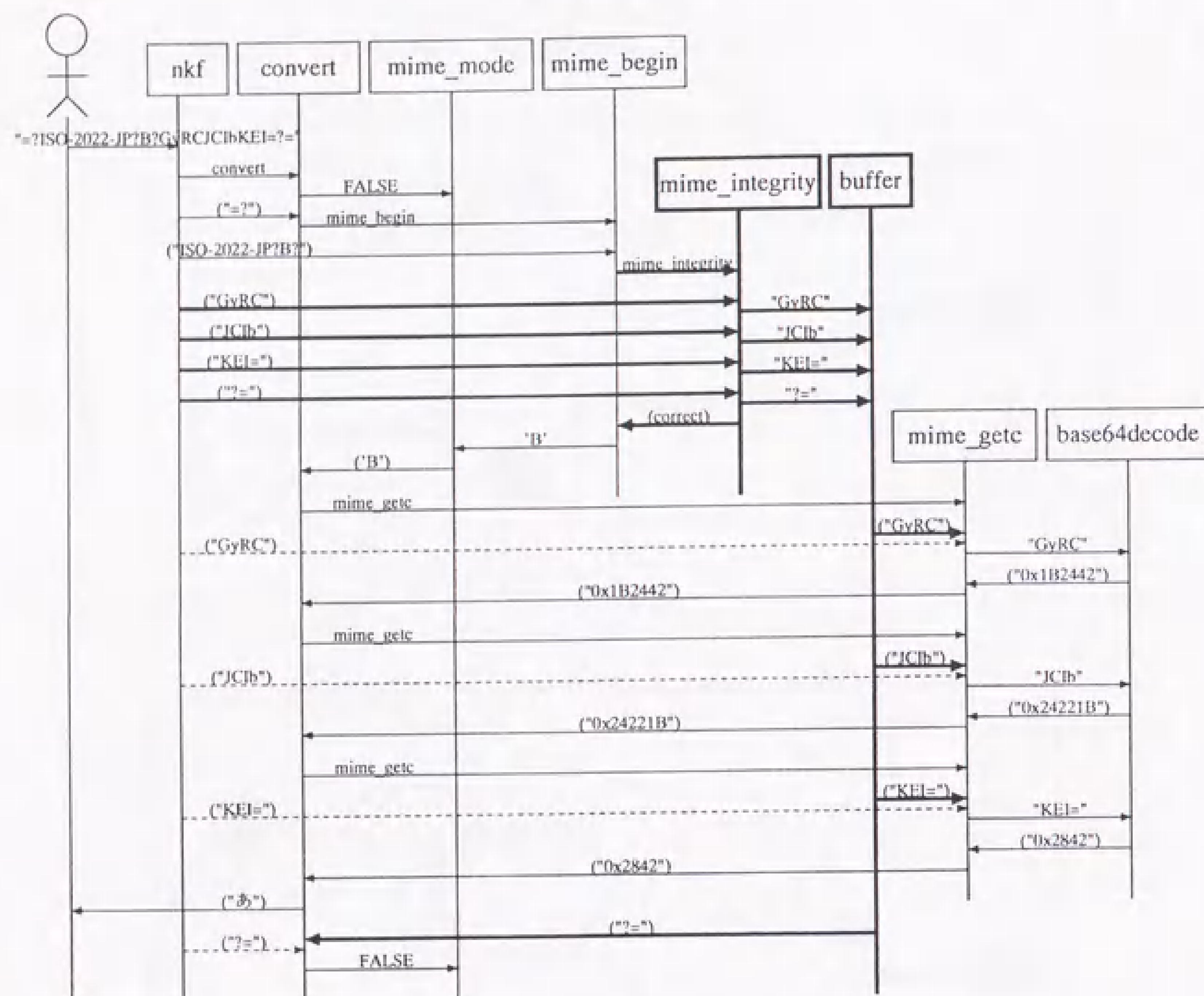


図 5.9: 新システムの入力 "=?ISO-2022-JP?B?GyRCJCIbKEI=?=" に対するシーケンス図

mime_integrity を追加し，関数 mime_begin() によって使用されていることを関数 mime_begin() との間の矢印によって表している．また，検査するために読み込まれた文字列は，復号化するために蓄積しておく必要があるため，buffer を用意した．これにより，関数 mime_getc() は，buffer から文字列を読み込むように変更されている．

5.3.3 機能の実現

5.3.2 節で記述した新システムの仕様を基に，既存システムから新システムへ変更する．5.3.2 節において追加した，読み込まれた文字列を蓄積しておくための buffer は，nkf1.5 において既に Fifo という形で実現されていたため，これを利用することにする．Fifo を利用するように変更したシーケンス図を図 5.10 に示す．

図 5.8 のステートチャートと図 5.10 のシーケンス図から，入力ストリームが正しいコードから構成されているかどうかを検査する関数 mime_integrity() を作成する．関数 mime_integrity() は，関数 mime_begin() から呼ばれ，内部で Fifo を利用している．

nkf は C 言語の 1 つのソースファイルから構成されているシステムである．そこで，MIME に関する処理の部分を別のファイルに分割することを考える．既にあるファイルに存在している関数を別のファイルに移動するためには，移動しようとしている関数とその関数を利用している関数の依存関係を考え，移動しようとしている関数に必要な宣言などとともに一緒に移動する必要がある．Sapid では，このような機能を関数スライシングツールとして実現している．この関数スライシングツールを利用することによって関数間の依存関係を考慮しながら適切にファイルを分割することができる．

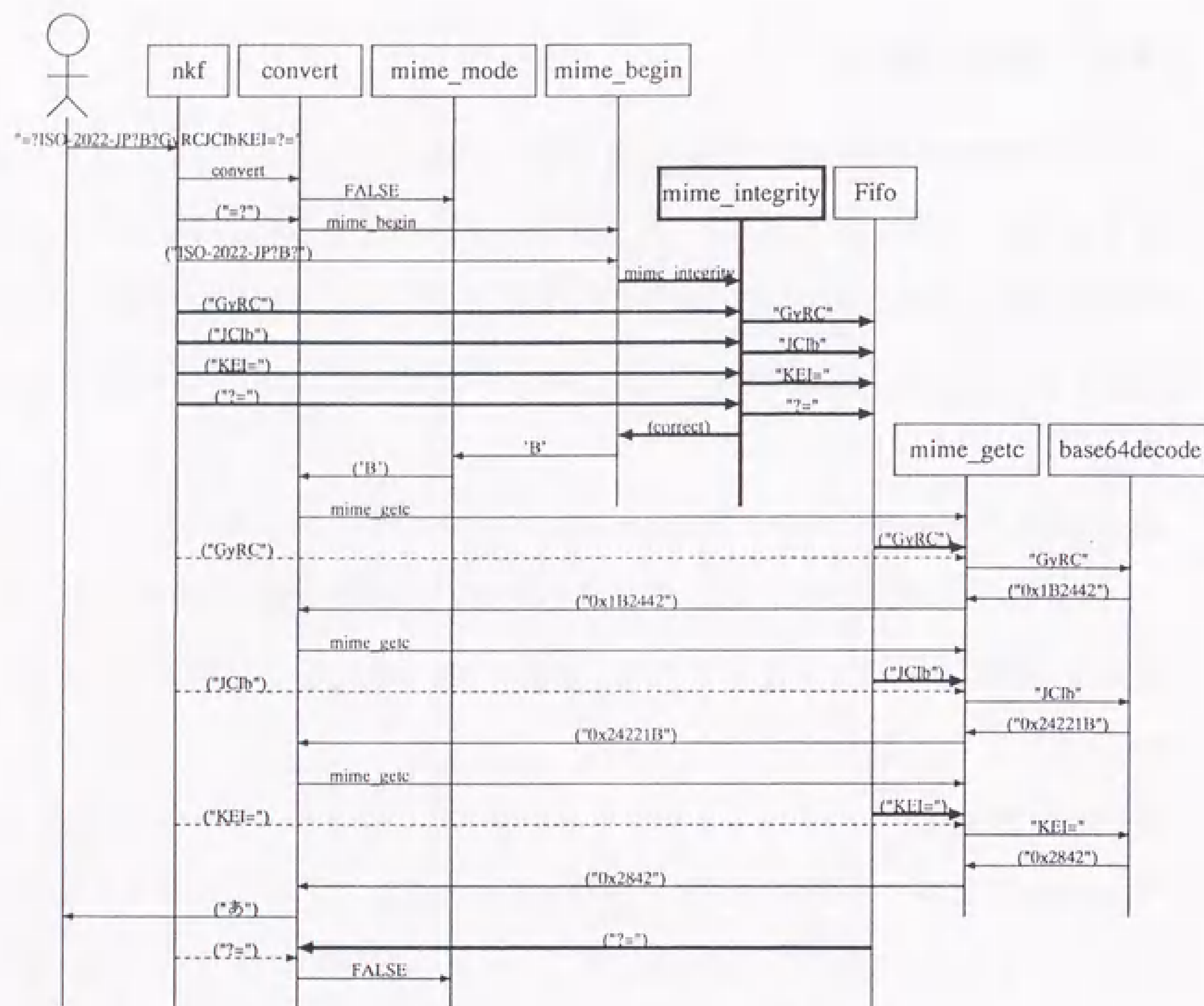


図 5.10: Fifo を利用するように変更したシーケンス図

5.4 再利用を支援するツール

5.3 節で述べた二つのツール，SPIE と関数スライサについて説明する．SPIE はタグ付けされたソースファイルを HTML などの形式で出力するツールである．SPIE は主に既存システムのソースプログラムを理解するために利用される．この解析結果に基づいて，既存システムの仕様を記述する．関数スライサは，ある関数を指定すると，その関数とともに，その関数に必要な宣言やマクロの定義などを抽出するためのツールである．関数スライサは，既存システムからある部品を取り出したり，関連のある関数群をまとめて別の一つのファイルにする際に利用される．

5.4.1 SPIE

既存システムから新システムへの要求に対して，新システムへの変更を意識しながら既存システムの仕様を記述するためには，既存システムのソースプログラムを理解することが重要である．この際，ソースプログラムのクロスリファレンスを作成することは，そのソースプログラムを理解するために有効である．SPIE [21] は，関数定義とその仕様の関係などを表現するためのアンカータグをソースプログラムに埋め込んだ HTML ファイルを生成するためのツールである．ここで HTML を利用した理由は，HTML とそのブラウザが広く利用されており，HTML のアンカータグが関数定義とその仕様などの関係を表現するのに適しているからである．SPIE は，次のような HTML ファイルを出力する．

- タグ付けされたソースプログラム
- ソースプログラムで定義されている関数の仕様
- 大域変数の表
- マクロの表

- ソースプログラムで定義されている構造体の表
- ソースプログラムで定義されている型の表
- ソースプログラムで使用されている定数の表

これらの HTML ファイルはアンカータグを用いて互いに結び付けられている．例えば，SPIE を用いることによって次のようなことが参照可能である．

- ソースプログラムのある関数定義からその関数の仕様の参照
- ソースプログラムのある関数呼び出しからその関数定義の参照
- ソースプログラム中の大域変数，マクロ，構造体のメンバ，型，定数の使用から，それぞれの表への参照

また，SPIE は関数仕様を LaTeX のソースファイルとして出力することも可能である．SPIE はソースプログラムの解析のために約 6,000 行，またタグ付けされた HTML ファイルの出力のために約 400 行の C 言語によって記述されている．

図 5.11 は SPIE の出力例である．右側のフレームには，関数名や変数名の部分にタグ付けされたソースプログラムが出力されている．ここで，init_stack() という関数定義の関数名の部分を選択すると，図 5.12 のように関数 init_stack() の仕様が表示される．図 5.12 によると，関数 init_stack() は，大域変数 num，stack，およびマクロ STACK_SIZE が使用している．また，関数 init_stack() は，関数 main() から呼び出されている．SPIE では，関数の他に，大域変数やマクロ，型などについても，同じように相互に関連付けられ自由に参照することができる．

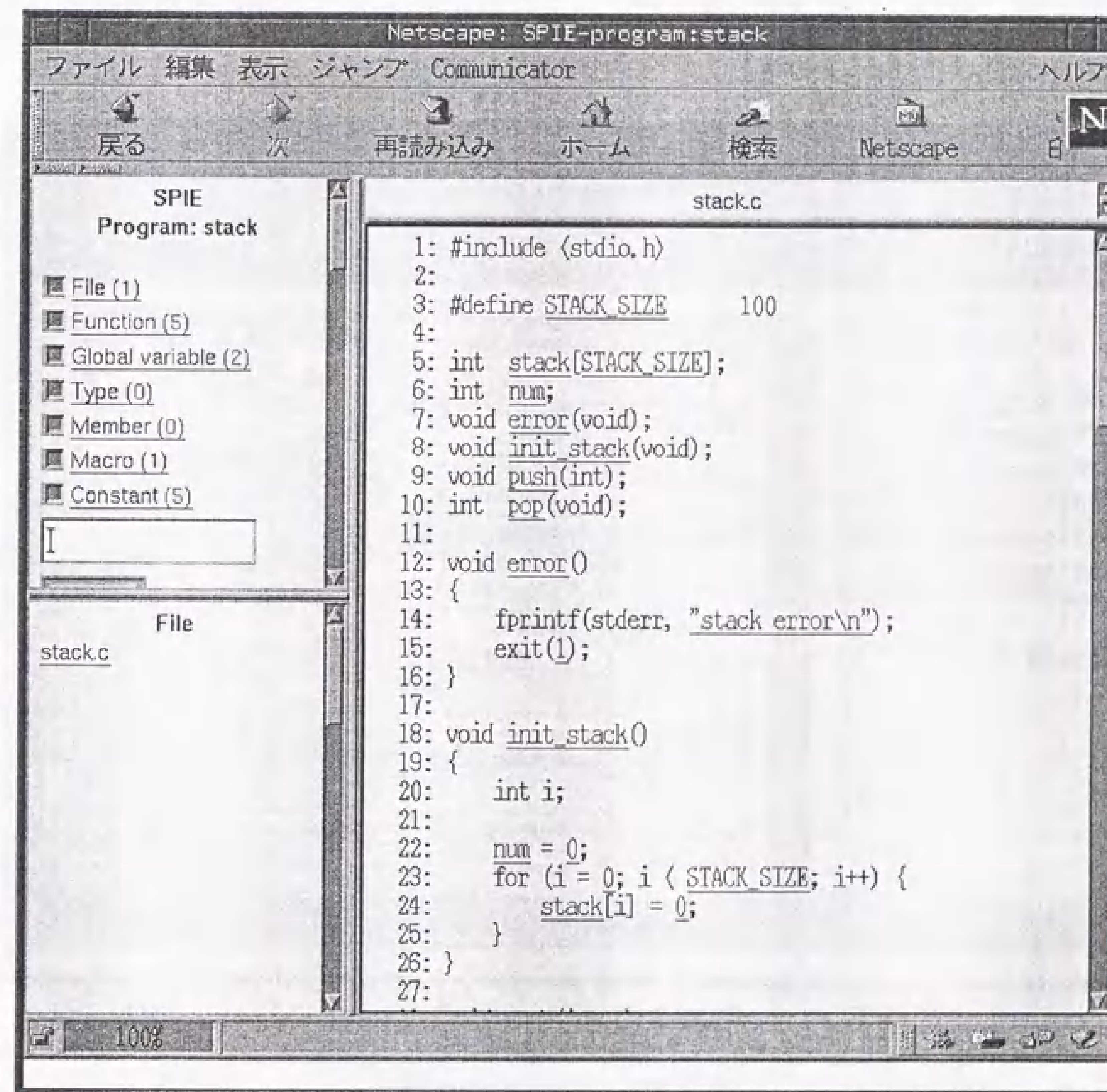


図 5.11: SPIE の出力例 1

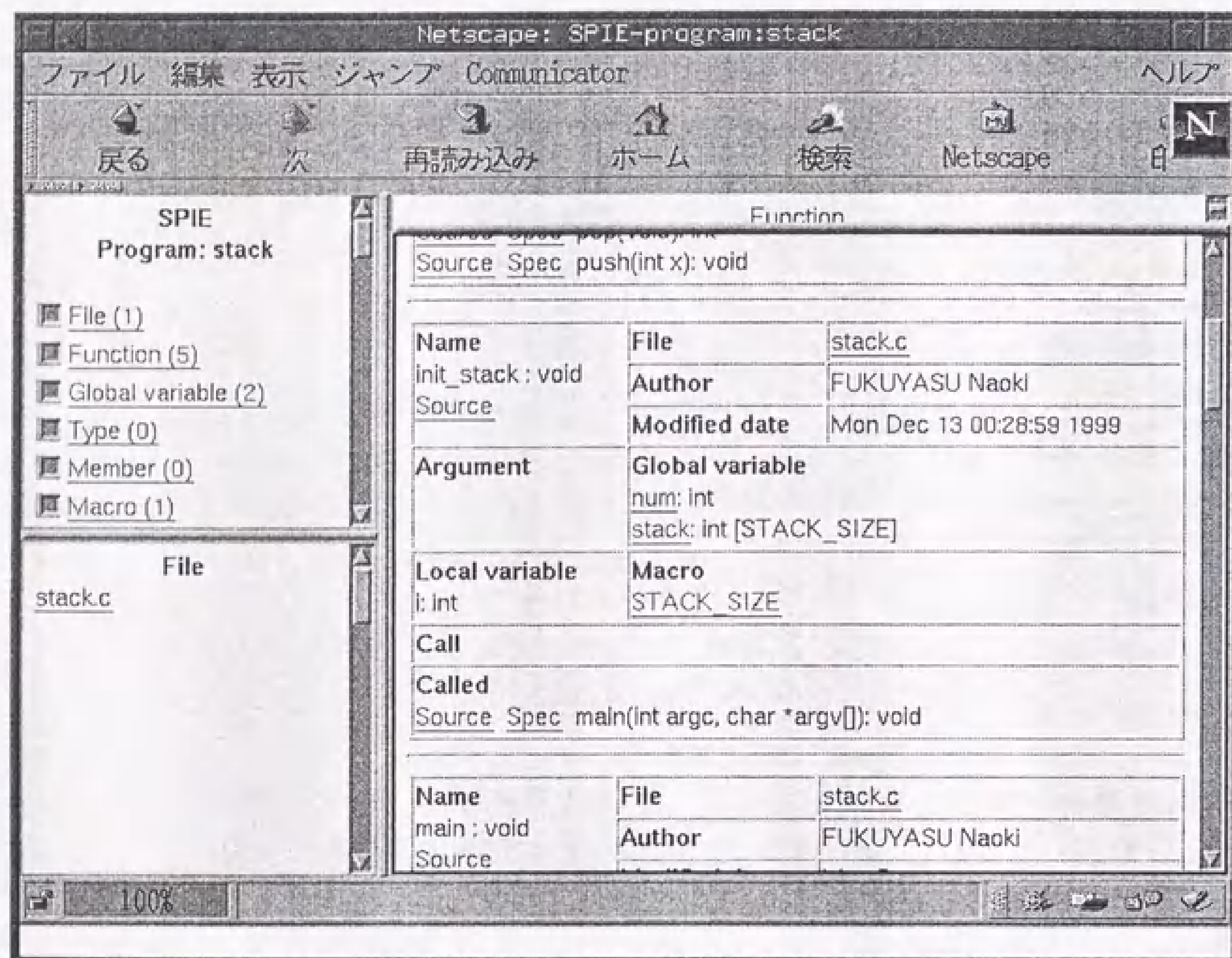


図 5.12: SPIE の出力例 2

5.4.2 関数スライサ

既存システムを再構成する目的で，ある処理をするための関数群を別の一つのファイルにまとめたいということがある．この際，既にあるファイルに存在している関数を別のファイルに移動するためには，移動しようとしている関数とその関数を利用している関数の依存関係を考え，移動しようとしている関数に必要な宣言などを一緒に別のファイルに移動する必要がある．関数スライサは，このようにある関数に対して，その関数と，その関数に必要な様々な宣言を一緒に抽出するためのツールである [12]．ある関数 f に注目した場合，関数スライサは，関数 f と関数 f で使用されている全ての識別子 i の宣言を抽出する．ここで，識別子 i は関数名，変数名，型名，およびマクロ名である．

関数 f が指定された場合，関数スライサは，

- 関数 f の定義とそのプロトタイプ宣言
- 関数 f から呼び出されている関数 f' の定義とそのプロトタイプ宣言
- 関数 f 内で定義・参照されている大域変数の宣言
- 関数 f 内で使用されている型の宣言
- 関数 f 内で使用されているマクロの定義

を抽出する．このツールは，約 1,600 行の C 言語のソースプログラムからなる．

図 5.13 に示すスタックを用いたプログラムから，関数 push() に関してスライスした結果を図 5.14 に示す．関数 push() では，大域変数 num， stack，マクロ STACK_SIZE，および関数 error() が使用されている．また，関数 error() では，stdio.h や stdlib.h の中で宣言されている関数 fprintf()， exit() などが使用されているため，関数 push() を抽出すると同時に，これらの宣言などについても抽出されている．

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #define STACK_SIZE      100
 4
 5 int  stack[STACK_SIZE];
 6 int  num;
 7 void error(void);
 8 void init_stack(void);
 9 void push(int);
10 int  pop(void);
11
12 void error()
13 {
14     fprintf(stderr, "stack error\n");
15     exit(EXIT_FAILURE);
16 }
17
18 void init_stack()
19 {
20     int i;
21
22     num = 0;
23     for (i = 0; i < STACK_SIZE; i++) {
24         stack[i] = 0;
25     }
26 }
27
28 void push(int x)
29 {
30     if (num >= STACK_SIZE) {
31         error();
32     } else {
33         stack[num] = x;
34         num++;
35     }
36 }
37
38 int pop()
39 {
40     if (num <= 0) {
41         error();
42     } else {
43         num--;
44         return stack[num];
45     }
46 }
47
48 void main(int argc, char *argv[])
49 {
50     int i, j;
51     init_stack();
52     j = 0;
53     for (i = 1; i < argc; i++) {
54         int k;
55         k = atoi(argv[i]);
56         if (k <= 0) continue;
57         push(k);
58         j++;
59         if (j > 5) break;
60     }
61     for (i = 0; i < j; i++) {
62         printf("pop: %d\n", pop());
63     }
64 }
```

図 5.13: 対象プログラム


```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #define STACK_SIZE      100
 4
 5 int  stack[STACK_SIZE];
 6 int  num;
 7 void error(void);
 8 void push(int);
 9
10 void error()
11 {
12     fprintf(stderr, "stack error\n");
13     exit(EXIT_FAILURE);
14 }
15
16 void push(int x)
17 {
18     if (num >= STACK_SIZE) {
19         error();
20     } else {
21         stack[num] = x;
22         num++;
23     }
24 }
```

図 5.14: 関数 push() に関するスライス

5.5　まとめ

ある既存のシステムに対して，環境の変化などから，新システムへの変更の必要が生じた場合，新しいシステムを作成する方法としては，新たに一から作成する方法と，既存システムに変更を加えることによって新システムへ進化させる方法の二つの方法が考えられる．ここでは，後者の既存システムに変更を加えることによって新システムへ進化させる方法に着目し，既存システムを再利用して新システムを作成するための枠組みを提案した．この枠組みは主に三つのステップから構成される．

1. 既存システムからの要求の抽出
2. 新システムの仕様の記述
3. 新システムの実現

まず，システムに対する不満を基に既存システムを解析することによって，新システムに対する要求を抽出する．既存システムに対するシナリオと，新システムに対するシナリオとの差分が，ユーザが抱いている不満に対する要求である．この要求を基に既存システムを解析して，新システムの仕様を既存システムの仕様からの差分という形で記述する．最後に，仕様の差分を基に既存システムに変更を加えるなどして新システムを実現する．仕様の記述には UML を用い，シナリオを中心に分析することによって再利用の可能な部分，および変更の必要な部分を見つける．

また，この枠組みにしたがって，nkf の MIME のチェック機能を強化する例を示した．さらに，この際に用いた Sapid の二つのツールについて説明した．SPIE は，既存システムのソースプログラムを理解するためのツールであり，関数定義とその仕様の関係などを表現するためのアンカータグをソースプログラ

ムに埋め込んだ HTML ファイルを生成する．関数スライサは，ある関数に対して，その関数と，その関数に必要な様々な宣言を一緒に抽出するためのツールである．提案した枠組みの中では，SPIE は主に既存システムを解析し新システムへの変更を想像するために用いられ，関数スライサは主に新システムへ変更する際のシステムの再構成に用いられる．Sapid は SPIE や関数スライサが必要とする情報を提供するのに十分細かい粒度であり，提案した枠組みの各フェイズにおいて必要な情報を簡単に取得することができる．

第 6 章

結論

ソフトウェアの再利用は，システムの開発や保守にかかるコストを削減し，同時にソフトウェアの品質を向上させる．既存のシステムを再利用することによって，ソースコードの記述に関してだけでなく，再利用した部分に対する分析，設計，検査の各段階においてもコストが削減される．また，実際の使用という形で十分にテストされたコードを利用することにより，ソフトウェアの信頼性が向上する．しかし，一般に既存のシステムから再利用可能な部分を見つけることは困難であり，また，見つかったとしても，通常そのままの形で再利用できることは少ないなどの問題がある．そこで，ソフトウェアの再利用を支援することが必要である．

これまでに，ソフトウェアの生産性向上を目指して，ソフトウェアの再利用だけでなく，ソフトウェアライフサイクルの各局面を支援する様々なCASEツールが開発されているが，一般にこれらのCASEツールは，ソフトウェアライフサイクルを全体的に支援するものではなく，個々の段階を支援する．しかし，要求分析から保守にいたるすべての局面が統一的に支援されなければ，十分な効率化は達成されない．

本論文では，要求分析から保守にいたるソフトウェアライフサイクルのすべての局面を統一的に支援することを目指して，細粒度の構成要素を対象とした

CASEツール・プラットフォームについて考察した．ソフトウェアライフサイクルの特に設計以降の局面では，ソースコードの字句要素や構文要素などを扱う作業が大きな割合を占めるため，ソフトウェアライフサイクルのあらゆる局面で生じる様々な問合せに応えるためには，これら細粒度の要素を扱うことが必要である．提案したCASEツール・プラットフォームの中核となる細粒度リポジトリは，ソフトウェアを構成するあらゆる要素を統一的に管理し，それらの情報を必要に応じて容易に取り出す機能を持つ．

また，細粒度リポジトリに基づいたCASEツール・プラットフォームをC言語を対象として実装したSapidについて述べるとともに，Sapidを利用していくつかのCASEツールを作成し，Sapidが，ソフトウェアライフサイクルのあらゆる局面を支援するのに十分な粒度であることを示した．

次に，ソフトウェアの再利用に必要な，既存のソフトウェアを一貫性を保ちながら変更する技術について考察した．ソフトウェアを構成するプログラムに存在する，構文規則やスコープ規則などの様々な制約を考慮し，変更操作を見通し良く行うことについて考察し，三層モデルを提案した．

さらに，既存システムに対して機能の追加や変更，バグの修正を行うような状況に注目し，再利用の可能な部分や，変更の必要な部分を見つける方法について考察した．また，既存システムを再利用することによって，新しいシステムを開発するための枠組みを示した．

第2章では，まずモジュールよりも粒度の細かい，モジュールの内部構造やその構成要素などのデータを扱う，細粒度リポジトリについて考察した．また，ソフトウェアを構成する様々な粒度の要素を統一的に管理する細粒度リポジトリを中核に持つ，CASEツール・プラットフォームを提案した．CASEツール・プラットフォームは，各種のCASEツールにおいて共通して必要となる，字句解析，構文解析などの機能を提供し，CASEツールを見通し良く実現するため

の基盤となる．

第3章では，C言語を対象に，第2章で述べた，細粒度リポジトリに基づいたCASEツール・プラットフォームSapidを実装した．Sapidは，主にSDBとARから構成される．SDBはSapidの基盤であり，要求されたソフトウェアをソフトウェアモデルI-modelに基づいて解析し解析した情報を蓄積する．C言語を用いたソフトウェア開発では前処理が多用される．そこで，この前処理に対応するために，SDBはPIDBを含んでいる．PIDBは前処理で行われるマクロ展開や指定されたファイルの包含などの情報を蓄積する．ARはSDBにアクセスするためのAPI関数群であり，ARで用意されるのはCASEツール作成者にとって必要かつ基本的なものである．また，ソースプログラムの制御依存関係やデータ依存関係を扱うためのビューとしてSDAを用意した．さらに，Sapidを利用したCASEツールを示し，Sapidが提供するAPIを用いて各種のCASEツールを見通し良く作成できることを確認するとともに，Sapidが実用的な時間的，空間的効率で動作することを示した．

第4章では，ソースプログラムの変更に的を絞り，ユーザの視点および変更にともなう制約からソースプログラムの変更を考察し，それぞれが三つに分類できることを述べた．また，ユーザの視点および変更にともなう制約から，ソフトウェアを一貫性を保ちながら変更するためのソフトウェアモデルとして，三層モデルを提案した．三層モデルは，ソフトウェアを表現層，構文層，記号層の三層によって表現するソフトウェアモデルである．この三層モデルにおいて，各層で三つの制約を別々に扱い，またユーザが変更する際の視点を各層に分けることにより，一貫性を保ったソースプログラムの変更操作を見通し良く行うことができることを示した．

第5章では，既存システムを新システムに進化させる枠組みについて考察した．ある既存のシステムに対して，環境の変化などから，新しいシステムを作

成する必要が生じた場合，既存のシステムの仕様をUMLを用いて記述し，修正し，その差分から既存のシステムを変更して，新しい環境に適応したシステムを作成する．この枠組みは主に三つのステップから構成される．

1. 既存システムからの要求の抽出
2. 新システムの仕様の記述
3. 新システムの実現

まず，システムに対する不満を基に既存システムを解析することによって，新システムに対する要求を抽出する．既存システムに対するシナリオと，新システムに対するシナリオとの差分が，ユーザが抱いている不満に対する要求である．この要求を基に既存システムを解析して，新システムの仕様を既存システムの仕様からの差分という形で記述する．最後に，仕様の差分を基に既存システムに変更を加えるなどして新システムを実現する．仕様の記述にはUMLを用い，シナリオを中心に分析することによって再利用の可能な部分，および変更の必要な部分を見つける．また，この枠組みにしたがってnkfのMIMEのチェック機能を強化する例を示し，Sapidにより，この枠組みを支援するために必要な情報が得られることを示した．

第3章で述べたSapidは解析結果を単なるUNIXのファイルとして管理している．そのため，排他制御や障害回復などの一般的なデータベース管理システムが持つべき機能がSapidでは実現されていない．Sapidの実装に一般的なデータベース管理システムを用いることは今後の課題である．

また，本論文では，主にソースプログラムを対象として，その構成要素などの，細粒度リポジトリによる管理について考察した．第4章で議論したソフトウェアの変更においても，その対象はソースプログラムであった．一般にソフトウェアは，ソースプログラムだけでなく，関連する各種仕様書やマニュアル，構成

管理やテストに関する書類など，様々なドキュメントによって構成される．しかし，ソースプログラム以外のドキュメントは各開発現場ごとに書式が異なり，問題が複雑になるためこれらについては考慮しなかった．ソフトウェア再利用においても，ソースプログラムだけでなくその他の各種ドキュメントを再利用することが重要である [23]．ソフトウェアを構成する様々なドキュメントを統一的に管理することは今後の課題である．ソースプログラム以外の仕様書やマニュアルなどが統一的に管理されれば，第5章で述べた枠組みにおいて，既存システムの仕様の作成の際により高度な支援が可能になる．

第5章では，仕様の差分による変更の支援については，例を示しただけであった．既存システムの仕様と新システムの仕様の差分を表現する方法についても今後の課題である．この際，仕様の差分とシステムの変更との間の関係について考察することによって，既存システムから新システムへの進化を自動的に支援することが可能になると思われる．

謝辞

本論文は，名古屋大学大学院 阿草清滋教授の懇切丁寧なる御指導によって作成することができました．深く感謝の意を表します．名古屋大学大学院 渡邉豊英教授，同 坂部俊樹教授には御多忙の中，論文原稿をお読み頂き，論文の構成およびその内容に関して御助言頂きましたことに深く感謝致します．

愛知県立大学 山本晋一郎助教授には，本研究の全課程を通して様々な相談に応じて頂き，懇切丁寧なる御指導を頂きました．深く感謝致します．

豊橋技術科学大学 吉田敦助手には，第4章で述べた三層モデルの作成にあたって数々の御助言を頂きました．深く感謝致します．

和歌山大学 鰺坂恒夫教授，大阪大学 井上克郎教授には，Sapid を実際に使用して頂き，システムの作成にあたって議論を重ね様々な御助言を頂きました．ここに深く感謝致します．

名古屋大学情報メディア教育センター 結縁祥治助教授，同 濱口毅助手をはじめ，名古屋大学阿草研究室の諸氏には，日頃より熱心に御議論頂き有益な御意見を頂きましたことに感謝致します．また，本研究は Sapid プロジェクトに関わられたすべての皆様の御力添えなくしては決して遂行することはできませんでした．心より御礼申し上げます．

最後に，これまで私を温かく見守って下さった両親に感謝致します．

参考文献

[1] 有賀寛朗, 山本晋一郎, 阿草清滋. ソフトウェア構造解析情報に基づくツールプラットホームシステム. 電子情報通信学会技術研究報告, Vol. SS94, No. 15, pp. 25–32, 1994.

[2] J. N. Buxon. *DoD Requirements for Ada Programming Support Environments.* United States Department of Defense, 1980.

[3] M. Chen and R. J. Norman. A Framework for Integrated CASE. *IEEE Software*, Vol. 9, No. 2, pp. 18–22, 1992.

[4] Y. Chen, M. Y. Nishimoto and C. V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, Vol. 16, No. 3, pp. 325–334, 1990.

[5] G. Engels, R. Huecking, S. Sauer and A. Wagner. UML Collaboration Diagrams and Their Transformation to Java. In *Proceedings of UML '99*, pp. 473–488, 1999.

[6] M. Fowler et al. *UML Distilled.* Addison Wesley, 1997.

[7] M. Gibbons. A Framework for Standardisation and Support Environment Technology. *Software Engineering Environments*, pp. 155–167, 1991.

[8] 蜂巣吉成, 山本晋一郎, 濱口毅, 阿草清滋. Java 言語のための細粒度リポジトリ. 情報処理学会 コンピュータシステムシンポジウム論文集, pp. 147–154, 1996.

[9] 羽生田栄一. オブジェクト指向モデリング言語 UML(1). 情報処理, Vol. 40, No. 1, pp. 76–82, 1999.

[10] 橋本靖, 山本晋一郎, 阿草清滋. Program Slicing を利用したプログラムカスタマイザ. 電子情報通信学会技術研究報告, Vol. SS94, No. 10, pp. 73–80, 1994.

[11] P. Haumer, K. Pohl and K. Weidenhaupt. Requirements Elicitation and Validation with Real World Scenes. *IEEE Transactions on Software Engineering*, Vol. 24, No. 12, pp. 1036–1054, 1998.

[12] 岩本奈美, 山本晋一郎, 阿草清滋. 関数スライサによるプログラム部品抽出手法とその応用. 情報処理学会 ソフトウェア工学研究会, Vol. 115, No. 4, pp. 25–32, 1997.

[13] I. Jacobson. *Object-Oriented Software Engineering.* Addison Wesley, 1992.

[14] S. C. Johnson. YACC — Yet Another Compiler Compiler. Computer Science Technical Report 32, Bell Telephone Laboratories, 1975.

[15] 川辺敬子, 上原三八, 金谷延幸. COBOL アナライザ: リバースエンジニアリング・プラットフォームの開発. 電子情報通信学会 知能ソフトウェア研究会, Vol. KBSE94, No. 32, pp. 1–8, 1994.

[16] M. E. Lesk. LEX — Lexical Analyzer Generator. Computer Science Technical Report 39, Bell Telephone Laboratories, 1975.

[17] 馬渕謙, 山本晋一郎, 酒井正彦, 阿草清滋. ソフトウェア保守におけるプログラム理解支援システム. 情報処理学会第43回全国大会講演論文集(5), pp. 253–254, 1991.

[18] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, pp. 308–320, 1976.

[19] 三村俊彦, 山本晋一郎, 阿草清滋. 前処理情報データベース. 電気関係学会東海支部連合大会講演論文集, p. 292, 1993.

[20] 小田章夫, 鯵坂恒夫. C プログラムに対するカプセル発見手法とその支援ツール. 電子情報通信学会論文誌, Vol. J79-D-I, No. 10, pp. 745–758, 1996.

[21] 大橋洋貴, 山本晋一郎, 阿草清滋. ハイパーテキストに基づいたソースプログラム・レビュー支援ツール. 電子情報通信学会技術研究報告, Vol. SS98, No. 28, pp. 15–22, 1995.

[22] R. Prieto-Diaz. Making Software Reuse Work: An Implementation Model. *ACM SIGSOFT Software Engineering Notes*, Vol. 16, No. 3, pp. 61–68, 1991.

[23] R. Prieto-Diaz. Software Reusability. *IEEE Software*, Vol. 10, No. 3, pp. 61–66, 1993.

[24] Reasoning Systems, Inc. *Refine User's Guide*, 1989.

[25] Sapid Home Page. `http://www.sapid.org/`.

[26] D. Shefström and G. van den Broek. *Tool Integration: Environments and Frameworks*. Wiley Professional Computing, 1993.

[27] J. L. Sourrouille. UML Behavior: Inheritance and Implementation in Current Object-Oriented Languages. In *Proceedings of UML '99*, pp. 457–472, 1999.

[28] 高田智規, 佐藤慎一, 飯田元, 井上克郎. ソースコード解析ツール開発支援システムの試用. 電子情報通信学会論文誌, Vol. J80-D-I, No. 3, pp. 317–318, 1997.

[29] G. Tatge. The Toaster Model, 1989.

[30] I. Thomas and B. A. Nejmeh. Definition of Tool Integration for Environments. *IEEE Software*, Vol. 9, No. 2, pp. 29–35, 1992.

[31] 内山晃司, 山本晋一郎, 阿草清滋. 依存関係が定義可能なテストベッド. 情報処理学会 ソフトウェア工学研究会, Vol. 106, No. 6, pp. 41–47, 1995.

[32] Unified Modeling Language. `http://www.rational.com/uml/`.

[33] L. Wakeman and J. Jowett. *PCTE The Standard for Open Repositories — Foundation for Software Engineering Environments*. Prentice Hall, 1993.

[34] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, pp. 352–357, 1984.

[35] 山本晋一郎, 阿草清滋. 細粒度リポジトリに基づいたツール・プラットフォームとその応用. 情報処理学会 ソフトウェア工学研究会, Vol. 102, No. 7, pp. 37–42, 1995.

[36] 吉田敦, 山本晋一郎, 阿草清滋. CASE ツール開発のためのソフトウェア操作言語. 情報処理学会論文誌, Vol. 36, No. 10, pp. 2433–2441, 1995.

[37] 吉田敦, 山本晋一郎, 阿草清滋. 意味を考慮した差分抽出ツール. 情報処理学会論文誌, Vol. 38, No. 6, pp. 1163–1171, 1997.

発表文献

論文誌

[38] 福安直樹, 山本晋一郎, 阿草清滋. 細粒度リポジトリに基づいた CASE ツール・プラットフォーム Sapid. 情報処理学会論文誌, Vol. 39, No. 6, pp. 1990–1998, 1998.

[39] 福安直樹，吉田敦，山本晋一郎，阿草清滋. 細粒度ソフトウェア・リポジトリに基づいたソースプログラムの安全な変更. コンピュータソフトウェア, Vol. 15, No. 4, pp. 78–81, 1998.

国際会議

[40] N. Fukuyasu, S. Yamamoto and K. Agusa. An Evolution Framework based on Fine Grained Repository. *Proceedings of International Workshop on Principles of Software Evolution(IWPSE99)*, pp. 43–47, 1999.

研究会 / シンポジウム

[41] 福安直樹，山本晋一郎，濱口毅，阿草清滋. 新しいソフトウェアモデルの提案. 電気関係学会東海支部連合大会講演論文集, p. 283, 1996.

[42] 福安直樹，山本晋一郎，阿草清滋. 細粒度ソフトウェア・リポジトリに基づいたソースプログラムの安全な変更. 日本ソフトウェア科学会第 14 回大会論文集, pp. 601–604, 1997.