CASEツール開発のためのソフトウェア操作言語

吉田 敦[†] 山本 晋一郎[†] 阿草 清滋[†]

本論文では，ソフトウェアに対する操作を直観的かつ簡潔に記述する言語として「ソフトウェア操作言語」を提案する．ソフトウェアは書式に従って形式的に記述された文書から構成されている．ソフトウェア操作言語では文書をオブジェクトとして，ソフトウェアに対する操作をオブジェクトに対するメソッドの適用としてとらえることで，直観的にわかりやすく操作を記述できる．また，書式をクラスとして扱うことで，操作に対して書式が与える制約を自然に扱うことができる．本論文では，特に文法として書式が明確に定義されているソースプログラムを操作対象として，ソフトウェア操作言語の概念とその記述方法を説明する．最後に，操作が簡潔に記述できることを表す例を示す．CASEツールはソフトウェアに対する操作の記述であるため，ソフトウェア操作言語を用いることで，CASEツールの開発が容易になる．本論文の手法は下流工程のCASEツールの開発が対象となるが，上流工程のCASEツールの開発についても適用可能であり，上流工程への応用が期待できる．

A Software Manipulating Language for CASE Tool Development

ATSUSHI YOSHIDA,[†] SHINICHIROU YAMAMOTO[†] and KIYOSHI AGUSA[†]

In this paper, we propose a language, called Software Manipulating Language, which enables us to describe the manipulation of software directly. We show the concept of the language and the way to describe manipulation, and then we give an example to show the effectiveness of the language. Software consists of the documents which generated from formats. Since we treat documents as objects and manipulation as methods, we can describe manipulation as application of methods to objects, which suits to intuition. We can also treat the restriction, which the formats impose on manipulation, by describing the formats as classes. The language encourages us to develop CASE tools, which are descriptions of manipulation. Though our main targets of manipulation are source programs, whose formats are given strictly, we can expect that our method are applicable not only to lower CASE tools, but to upper CASE tools.

1. はじめに

ソフトウェアの開発や保守・管理においてCASEツールは不可欠である．特に，上流工程における作業を下流工程に自動的あるいは半自動的に反映させることで生産性や保守性が向上するため，上流工程のCASEツールについて多くの研究や開発が行われている．

一方，上流工程を支援するためには，下流工程の支援が不可欠である．特に，上流工程における作業を下流工程に反映させるためには，下流工程の作業を十分に支援できるCASEツール群や，それらを含んだ環境が必要となる．また，ソフトウェアの再利用や保守管理など，下流工程を中心とする作業の支援も十分とは言えず，下流工程の支援は不可欠である．

しかし，現実には，下流工程を十分に支援できる環境が整っていないにも関わらず，多くの研究が上流工程の支援に重点を置いている．下流工程はソフトウェアのライフサイクルの中でも大きな割合を示めるため，まず，下流工程を十分に支援することが必要である．

我々は主に下流工程を支援するために，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（Sophisticated Application Programming Interface for software Development）を開発している[1]．細粒度リポジトリとは，従来のリポジトリによるデータ統合や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において[2),3)]，統合化の対象とはならなかった細粒度の構成要素を対象とするリポジトリである．従来のリポジトリを用いた場合，最小単位がモジュールであるため，ファイルよりも細かい単位の構成管理，ソフトウェアの部品化・再利用などの作業が困難であるが，細粒度リポジトリでは関数や変数などの粒度の細かい成果物を対象とした作業を支援で

† 名古屋大学工学部
School of Engineering, Nagoya University

きる．

細粒度リポジトリによって，従来は各CASEツールが個別に持っていた解析器や，各CASEツールに共通する機能が提供されるため，CASEツール開発者はCASEツールの本質的な機能の実現に専念できる．

CASEツールとは，ソフトウェアの開発過程で頻繁に生じる定型作業を記述したものである．その定型作業は，ソフトウェアの解析情報を得る操作やソフトウェアに対する変更操作からなる．CASEツール開発を支援するためには，このようなソフトウェアに対する操作を記述する枠組が必要である．本論文では，ソフトウェアに対する操作の直観的かつ簡潔な記述のための言語として，ソフトウェア操作言語を提案する．ソフトウェア操作言語は，細粒度リポジトリに対するアクセスを容易にし，CASEツールの構築を容易にする．

2. ソフトウェアに対する操作

2.1 ソフトウェア

ソフトウェアとは文書であり，一般に，各種仕様書，構成管理文書，ソースプログラム，実行形式ファイルなどの様々な文書から構成される．これらの文書は書式に従って形式的に記述されている．本論文では，これらの文書のうち，特に重要であるソースプログラムを操作の対象とする．その他の文書は開発現場ごとに書式が異なり，統一的な扱いが難しいため，本論文では扱わない．ただし，ソースプログラム以外の文書についても，書式が明確に規定され，解析が可能であれば，本論文の手法は適用可能である．そのため，本論文で提案する言語の名前には「ソースプログラム」の代わりに「ソフトウェア」を用いている．以下では，ソースプログラムに対象を絞って説明をする．

なお，ソースプログラムの書式とは文法であり，構文規則と意味規則からなる．ただし，実際の開発現場においては，ソースプログラムの記述方法について文法以外の規定が存在する場合があり，その場合にはその規定も書式に含むため，「文法」の代わりに「書式」を用いる．

2.2 操　　作

ソースプログラムに対する操作とは，ソースプログラムを解析して情報を得ることと，それらの情報を基に加工することである．ここで，解析とは依存解析や波及解析などであり，解析から得る情報とは，program slicing[4]における特定の出力を得るための必要最低限のステートメントや，関数仕様書の作成に必要な各関数ごとの参照変数や関数呼び出し，マクロ呼び出しなどの情報である．また，ソースプログラムに対する加工とは，抽象実行[5]などによる最適化やInterface Adaptation[6]などによる再利用のための変更，あるいは仕様書との整合性を保った変更などである．

2.3 CASEツール開発の問題点

ソースプログラムに対して操作するCASEツールとして，クロスリファレンサ，差分抽出ツールおよびプリティプリンタなどが実際に用いられている．これらのツールは，それら自身が解析器を持ち，それぞれの操作で必要な解析を行っている．このようなツールを作成する場合，必ず解析器が必要であるが，一般にソースプログラムの書式は複雑であるため，ツールを作成する際の作業量を大きくする要因となる．書式が変更されることは稀有であり，また，ツールが必要とする解析データの多くは共通するため，各ツールに共通な解析器を用意することは有効である．特に，ソースプログラムは書式が厳密に規定されているため，他の仕様書などの文書に比べ，解析器を容易に作成できる．解析器を用意することで，ツールを作成する際に，ツールに本質的な操作の実現のみに専念できる．

2.4 CASEツール・プラットフォームSapid

我々は，様々なCASEツールに共通な解析データを提供するCASEツール・プラットフォームとしてSapidを開発している．Sapidは，**図1**に示すように，ソフトウェアデータベース（SDB: Software Database），アクセスルーチン（AR: Access Routines），ソフトウェア操作言語（SML: Software Manipulating Language）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に基づいて解析する解析器（Analyzer）と，解析の結果から得られた実体・関連情報を格納するデータベースからなる．**図2**に示すよ

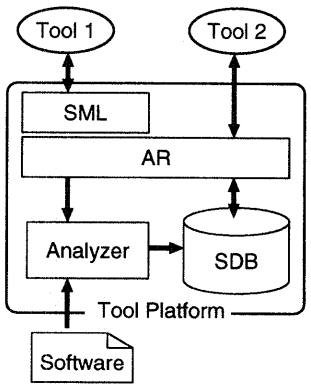


図1　ツールプラットフォームの構成
Fig. 1　System of tool platform.

うに，我々はC言語を13の実体と33の関連からなる実体関連モデルとしてとらえている．このモデルはC言語の文法に基づいて作成されたが，ツール開発者の立場から必要となる実体や関連の付加や，不要な実体や関連の削除によって，モデルを簡潔で扱い易くしている．なお，本論文で示す記述例はこのモデルに基づくが，C言語以外の手続き型言語に関しても同様に記述可能である．

ARはSDBに対するアクセス機能を提供する．メタデータ取得メソッド，データ属性値取得メソッド，関連取得メソッド，関連オブジェクト取得メソッドを持ち，C言語のライブラリとして提供される．ここで，メタデータとは，データベース内のデータを構成する実体名，関連名，および実体，関連の持つ属性名，属性の型を示す．

また，ARはソフトウェア開発において多用される前処理に対処するための前処理データベース（PIDB: Preprocess Information Database）を含む．SDBに対して前処理前の情報を持たせるとモデルが複雑になるため，SDBには前処理後の情報を，PIDBには前処理の前後の対応に関する情報を格納させることで，前処理に対処する．ツールからのアクセス要求は前処理後のソースプログラムに対するアクセス要求に変換される．SDBを検索して得られた情報は前処理前のソースプログラムの情報へ変換され，ツールに返される．

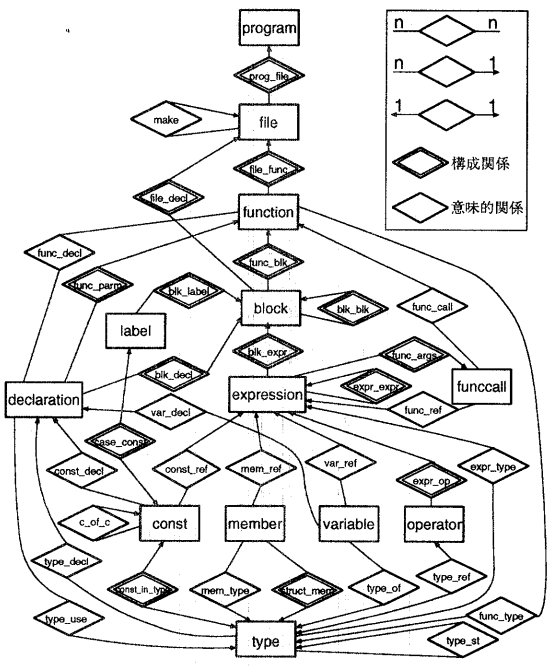


図2 C言語の実体関連モデル
Fig. 2 Software model.

CASEツール開発の支援を目指したシステムとしては，CIA[7]が挙げられる．CIAはC言語のソースプログラムから，ファイル，関数，外部変数，型，マクロに関する情報を抽出して関係データベースに蓄積し，ユーザは問い合わせ言語を用いてソースプログラムの内容に関する問い合わせをする．CIAは，クロスリファレンサに代表される情報抽出のためのシステムであり，ソースプログラムを変更する機能を持たないため，CASEツールプラットフォームとしては不十分である．

3. ソフトウェア操作言語

3.1 操作の記述

CASEツールは，ソフトウェア開発過程で頻繁に生じる定型作業を記述したものであり，ソフトウェアに対する操作の記述である．よって，CASEツール開発の支援には操作を直観的にわかりやすく記述できる言語が必要である．

Sapidでは，解析結果にアクセスするためにARがC言語のライブラリとして提供されており，C言語を用いて操作を記述することができる．しかし，ARは基本的なアクセス機能のみを提供するため，複雑なCASEツールを作成することは容易ではない．また，ソフトウェアに対する操作を記述する際には，以下に述べる制約を意識しなければならない．制約は操作に共通して存在することから，制約を取り扱うことは重要である．よって，簡潔に操作が記述可能で，かつ制約を自然に扱える言語が必要である．

3.2 制　　約

ソースプログラムを変更する場合，書式に従う必要がある．例えば，ソースプログラム内の変数を変更する場合，名前空間を考慮して，同じ識別子を持つ変数を区別する必要がある．さらに，関数仕様書とソースプログラムの間でも整合性を保たなければならない．

操作を行う際には，常に書式による制約が伴い，ソースプログラムの開発者は書式による制約を意識しながら操作する必要がある．しかし，エディタなどの一般的なツールを用いた開発では，書式は暗黙的に制約を与えるのみであるため，書式による制約を無視することが可能である．**図3**の左図はこの様子を示し，書式には制約が存在するが，ソースプログラムに対しては明示的な制約を加えない．よって，開発者が気づかずに不正な操作を行い，ソースプログラムに誤った変更を行う危険がある．このような危険を回避するためには，危険を伴う操作の記述を回避できる環境が必要である．

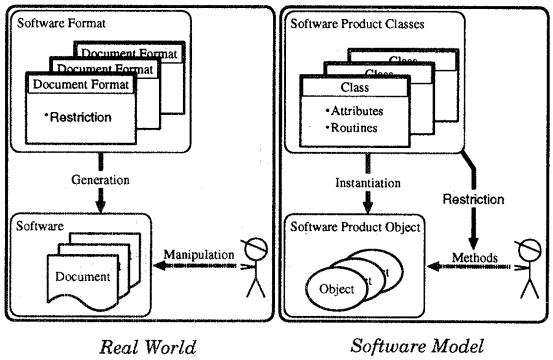


図3 書式による制約
Fig. 3 Restriction imposed by formats.

3.3 ソフトウェア操作言語の提案

ソフトウェア操作言語は，ソフトウェアの構成要素をオブジェクトとして扱い，構成要素の特性をクラスとして定義するオブジェクト指向言語である．このクラスを特に「書式クラス」と呼ぶ．定義される特性は，構成要素間の関係，名前や種類などの属性，基本操作であり，インスタンス変数やメソッドとして記述される．構成要素間の関係については構成関係と，依存関係などの意味的関係を区別して扱う．基本操作は，各構成要素に固有な操作であり，制約を満たすように定義される．この書式クラスは，細粒度リポジトリとして提供されるソースプログラムに対するビューであり，CASE ツール開発者はこのクラスを用いて操作を記述する．実際に操作を記述するクラスを「操作クラス」と呼び，各操作をメソッドとして記述する．

ソフトウェア操作言語は，特定の条件を満たすオブジェクトを探索する機能を持つ．これは，ソフトウェアに対する操作を行う場合，まず操作対象の特徴を考え，次にその特徴に基づいて操作対象を探索し操作を行うためである．探索は，集合の内包的記法によって記述され，その結果は集合となる．集合を用いる理由は，直観的にわかりやすく，またアルゴリズムの実現が容易になることである．実際，アルゴリズムを自然に実現するために，SETL[8]や SOL[9]などの集合指向言語が研究されている．また，「ある関数に含まれる変数」など，探索条件として構成関係が多く用いられる．このため，書式クラスでは構成関係とその他の意味的関係を区別して扱い，構成関係に基づく探索を可能とする．

オブジェクト指向技術を用いる理由は，直観的かつ簡潔な記述が可能になることと，制約を明示的に記述できることである．一般に，ソースプログラムに対して操作を行う場合，そのソースプログラムの開発者は変数や式，関数，ファイルなどの構成要素を単位として考える．よって，構成要素をオブジェクトとしてとらえることで，より直観的な記述が可能になる．さらに，書式で規定される構成要素以外が加えられる余地がないため，構成要素に関する制約を自然に満たす．また，各構成要素に対する基本的な操作は，操作内容に関わらず決まるため，各種類の構成要素の集合をクラスとみなし，クラスごとに基本操作をメソッドとして定義することで，基本操作を再利用できる．

一般に，オブジェクト指向言語では情報隠蔽の概念が採り入れられている．情報隠蔽を用いることで，各クラスに定義された操作のうち，制約を満たした操作のみを開発者に提供する．例えば，変数名を置換する操作の場合，変数名を直接変更することを禁止し，代わりに変数名を変更するメソッドを提供する．このとき，変数名変換に伴う他の変更や，変数名の衝突の回避などをそのメソッドに記述することで，制約を満たした操作のみが可能になる．図3の右図はこの様子を示し，クラスが明示的に操作に制約を加え，開発者はクラスに定義されたメソッドのみが利用可能である．

4. ソフトウェア操作言語の記述

前章で述べたソフトウェア操作言語の具体的な例として，Spaid において実際に採用している言語について示す．ソフトウェア操作言語の設計に当たっては，開発の容易さから，既存の言語を拡張することとし，オブジェクト指向言語 Sather[10]を採用した．

Sather は，大規模なアプリケーションを，簡潔で効率的かつ安全に開発することを目標としたオブジェクト指向言語である．Eiffel[11]から派生し，他の言語の技術も取り入れた独自の発展を遂げており，将来性がある．また，処理系のソースコードが公開されており，自由に使用できる．Sather を採用した理由は，構文が簡潔であること，将来性があること，静的かつ強い型検査があることである．静的かつ強い型検査は，制約を無視した危険な操作を実行前に防止できるため重要である．

Sather からの拡張は，構成要素間の関係を表すための予約語の追加と，集合に関する表記の追加である．追加した部分以外は構文を変更していないため，Sather のクラス・ライブラリも利用可能である．

ソフトウェア操作言語による記述は，書式に対するビューを表す書式クラスと，操作を記述する操作クラスから成る．以下では，拡張した部分を中心に各クラスについて説明をする．また，拡張した部分以外で，本論文を理解する上で必要がある事項については適

宣説明する．また，構文規則および各書式クラスで現時点で定義されている属性およびルーチンの一覧を付録に付加する．構文規則中で“(*)”を伴う記号は拡張によって追加された非終端記号を表す．定義一覧で“component”，“relative”が付いているものは属性を表し，それ以外はルーチンを表す．なお,「属性」とはインスタンス変数を,「ルーチン」とはメソッドを意味し，以下の説明で用いる．

4.1 書式クラスの記述

4.1.1 関係の記述

一つの書式は一つのクラスとして定義される．例えば，ソースファイルや関数，式，変数などの書式をそれぞれクラスとして定義する．構成関係や意味的関係が属性として定義され，属性の値は各関係にあるオブジェクトまたはオブジェクトの集合になる．

書式には必ず構成関係が存在する．構成関係を意味的関係と区別するために，構成関係の属性は予約語 **component** または **components** の直後に定義される．また，意味的関係については，予約語 **relative**, **relatives** を用いる．それぞれの予約語は単数形と複数形が存在するが，単数形は，ただ一つのオブジェクトと関係することを表し，属性値は関係するオブジェクトである．複数形は，複数のオブジェクトと関係することを表し，属性値は関係するオブジェクトの集合である．なお，このときの集合は順序集合であり，順序関係は出現位置によって決まる．また，**component** や **relative** などの予約語はソフトウェア操作言語で新たに加えられた予約語であり，関係を表す属性以外の属性はこれらの予約語を付けずに宣言する．

例えば，関数に対するクラス **FUNCTION** を次のように定義する．

```
class FUNCTION is
  component  name:       ATTRIBUTE;
  components arguments:  DECLARATION;
  components blocks:     BLOCK;
  relative   a_type:     A_TYPE;
  relatives  callers:    FUNCTION_CALL;
end;  -- FUNCTION
```

ここでは，関数を構成する要素を関数名，引数，関数の本体ブロックとし，それぞれ，属性 **name**, **arguments**, **blocks** として定義する．関数はただ一つの名前を持つため **name** には単数形の予約語が用いられる．また，引数や，関数本体を構成するブロックは複数存在するため，**arguments**, **blocks** には複数形の予約語が用いられ，それぞれの型は宣言クラスの集合，ブロック・クラスの集合である．

我々のモデルでは，型や関数の呼出しも実体として扱い，それらのオブジェクトと関数オブジェクトは参照関係にあると考える．そのため，関数の型や関数呼出しを意味的関係として定義する．例えば，関数の型については属性 **a_type** として定義し，関数呼出しについては属性 **callers** として定義する．

4.1.2 基本操作の記述

各書式クラスに，制約を満たした基本操作をルーチンとして記述する．関係を表す属性に対する操作のために，基本的なアクセス・ルーチンが自動的に定義される．定義されるアクセス・ルーチンには，単に属性値の設定や取得を行うルーチンだけでなく，ソフトウェアに特有な基本操作を行うルーチンが含まれる．例えば，構成関係を表す集合にオブジェクトを追加するときは，オブジェクト間の順序関係を考慮する必要がある．よって，複数のオブジェクトとの構成関係を表す属性に対するアクセス・ルーチンとして，追加するオブジェクトと相対位置を引数に取る挿入ルーチン **insert** が定義される．また，操作対象のオブジェクトの取得や，オブジェクトの生成のために，すべての書式クラスにおいて基本ルーチン **target**, **create** が定義される．

基本ルーチンを組み合わせて操作を行う場合，実行順に制約を受ける場合がある．例えば，ある変数宣言を削除するためには，まずその変数の参照が削除される必要がある．先に変数宣言を削除した場合，宣言されていない変数を参照することになり，意味的に正しくない．このような制約に対しては，新たにルーチンを追加することで対処する．また，名前変え機能を用いて，ルーチンを制約を満たすように再定義して対処することも可能である．

4.2 操作クラスの記述

操作の記述は，オブジェクトに対してルーチンを実行する形式で記述する．以下では，ソフトウェア操作言語の特徴的な点として，操作対象のソフトウェアに対応するオブジェクトの取得方法，集合に関する記述方法について説明する．

4.2.1 オブジェクトの取得

一般に，操作を開始する前に操作対象の文書オブジェクトを取得する必要がある．このオブジェクトの取得には，ルーチン **target** を用いる．引数に操作対象の文書を指定してそのクラスのルーチンを呼び出すと，その文書に対応するオブジェクトが得られる．例えば，あるソースファイルに操作を行う場合は，次のように記述する．

```
file: SOURCE_FILE :=
          SOURCE_FILE::target(filename);
```

SOURCE_FILE はソースファイル・クラスを表し，filename はソースファイルの名前を値として持つ変数である．“::” はクラスのルーチンに対する呼び出しを意味し，ソースファイル・クラス SOURCE_FILE のルーチン target を呼び出すことで，target の引数に指定されたソースファイルに対応するオブジェクトが得られる．“:=” は代入を意味し，得られたソースファイル・オブジェクトはソースファイル・クラス型の変数 file に代入される．この変数 file を参照し，ルーチンを呼び出すことでソースファイル・オブジェクトに対する操作を行う．なお，ステートメント内の任意の箇所で局所変数を宣言することが可能であり，この例の変数 file もこのステートメント内で宣言された局所変数である．

4.2.2 集　　合

ソフトウェア操作言語では，操作対象の探索と探索して得られたオブジェクトに対する操作を簡潔に記述するために，(1) 内包形式による集合表記と (2) 集合要素ごとの繰り返し文の表記を用意した．なお，集合クラスによって集合を扱い，一般的な集合演算も可能である．

(1) 内包形式による集合表記

内包形式による集合表記は，操作対象の特徴を記述することで，その特徴を持つオブジェクトの集合を表す．記述方法を以下に示す．

```
{要素変数:クラス名 in 範囲式 | 論理式}
```

特徴は**クラス名**，**範囲式**，**論理式**によって表される．**クラス名** は要素の型を表し，**範囲式**は，探索を開始するオブジェクトを指定する．探索は，指定されたオブジェクトから構成関係を再帰的にたどり，その経路上のすべてのオブジェクトについて行われる．そのうち**論理式**を満たすオブジェクトが集合の要素となる．**論理式**の中では**要素変数**を参照することにより，集合の一要素を参照する．なお，**クラス名**と**論理式**はそれぞれ省略可能である．例えば，あるファイルに含まれる変数の中で，名前が “foo” である変数の集合は次のように記述する．

```
{var: VARIABLE in file
          |var.name_is("foo")}
```

変数 file は探索対象のファイル・オブジェクトを参照する変数とする．変数 var は変数クラス VARIABLE 型の変数で，集合の一要素として論理式の中で参照されている．name_is は名前が引数に与えられた文字列に等しいか否かを調べるルーチンであり，その結果を論理値として返す．

(2) 集合の要素ごとの繰り返し文

内包形式による集合表記などを用いてオブジェクトの集合を得た場合，その集合の各要素に対して同一の操作を繰り返すことが多い．この集合の要素ごとの繰り返しを次のように記述する．

```
foreach 変数名:クラス名 in 集合 do
  操作; ....; 操作
end
```

集合の中の任意の要素が**変数名**として宣言される変数により参照され，それぞれの要素に関して操作が繰り返される．この宣言される変数の型は**クラス名**として記述されるクラスであり，**集合**の要素の型はこのクラスまたはこのクラスのサブクラスでなければならない．例えば，あるソースプログラムの中で定義されている関数の名前の一覧を出力するためには次のように記述する．

```
foreach func: FUNCTION
          in {f: FUNCTION in file} do
  OUT::s(func.get_name.str).nl;
end
```

なお，ルーチン**get_name** はオブジェクトの名前オブジェクトを返し，ルーチン str によって文字列に変換される．OUT は標準出力のクラスであり，ルーチン s は引数の文字列を，ルーチン nl は改行をそれぞれ出力する．

5. 操作の記述例

操作言語の有効性を示す例として，次のようなソースプログラムの簡単化の例を示す．

簡単化操作: ソースプログラム内で定義されている関数ごとにその関数呼出しを調べ，すべての関数呼出しにおいて i 番目の実引数が同じ定数であれば，関数定義内の i 番目の仮引数の参照をその定数に置き換え，i 番目の実引数と仮引数宣言を削除する．□

なお，仮引数は参照されるのみとし，変更されないと仮定する．

図 4 に操作記述を示す．この記述ではクラス SIMPLIFY が定義され，その中でルーチン main と simplify_arguments が定義されている．ルーチン main はメインルーチンを表し，引数に指定されたファイル名からファイル・オブジェクトを取得し，そのオブジェクトの中の関数オブジェクトごとにルーチン simplify_arguments を呼び出す．最後に，ファイルのテキストを取り出して表示する．ルーチン simplify_arguments は引数に与えられた関数オブジェクトの関数呼出しオブジェクトを取得し，実引数に同じ定数が使われているか否かを調べる．

```
1 class SIMPLIFY is
2   -- Simplify Arguments
3
4   main(args: ARRAY{STR}) is
5     file: SOURCE_FILE := SOURCE_FILE::target(args[1]);
6
7     foreach f: FUNCTION in {f: FUNCTION in file} do
8       simplify_arguments(f);
9     end;  -- foreach
10
11    OUT::s(file.get_text);
12  end;  -- main
13
14  simplify_arguments(func: FUNCTION) is
15    num_of_args: INT := func.get_arguments.size;
16    args: ARRAY{EXPRESSION} := args.new(num_of_args);
17
18    -- looking for target arguments which satisfy the condition
19    first: BOOL := true;
20    foreach f:FUNCTION_CALL in func.get_callers do
21      arg_list: LIST{EXPRESSION} := f.get_arguments.to_list;
22      i: INT; until i = num_of_args loop
23        arg_i: EXPRESSION := arg_list.pop;
24        if first and arg_i.is_a_CONSTANT then
25          args[i] := arg_i;
26        elsif args[i] /= void and arg_i.is_a_CONSTANT then
27          if not arg_i.same(args[i]) then
28            args[i] := void;
29          end;
30        end;  -- if not first
31        i := i + 1;
32      end;  -- loop
33      first := false;
34    end;  -- foreach
35
36    -- replacement of references of formal arguments, and
37    -- elimination of actual arguments
38    i:INT; until i = num_of_args loop
39      if args[i] /= void then
40        func.get_an_argument_variable(i).replace_with(args[i]);
41        func.remove_an_argument(i);
42        foreach f: FUNCTION_CALL in func.get_callers do
43          f.remove_an_argument(i);
44        end;  -- foreach
45      end;  -- if
46      i := i + 1;
47    end;  -- loop
48  end;  -- simplify_arguments
49 end;  -- SIMPLIFY
```

図4 簡単化操作の記述
Fig. 4 Description of simplification.

実行例として**図5**に入力ソースプログラム，**図6**に出力ソースプログラムを示す．図5では，メイン関数の他に**add()**，**sub()**，**print()**が定義されている．これらの関数の呼出しを調べると，それぞれの関数呼出しに共通する実引数が存在する．例えば，関数**print()**の場合，第1引数の文字列定数**"%s(%d, %d) = %d\n"**と，第3引数の整数定数10が存在する．これを，変換すると図6になる．関数**print()**については，第1引数，第3引数が削除され，15行で参照していた仮引数が定数に置換されている．

この操作の記述量は49行と短く，ソフトウェア操作言語により簡潔な記述ができることを表している．集合に関する記述を用いた点と基本的かつ有効なルーチンが定義された書式クラスがライブラリとして用意されている点が簡潔化に大きく寄与している．

また，ソフトウェア操作言語の処理系のプロトタイプを作成した．その処理系はソフトウェア操作言語による記述からSatherによる記述へ変換し，Satherの処理系によりコンパイルする．この変換器はyaccのコードを含めてC言語で約3800行，Satherの処理系でコンパイルするのに必要なクラス記述が35クラス，284ルーチン，約3000行である．図6は，図4をこ

```
1 #include <stdio.h>
2
3 int add(int x, int y)
4 {
5     return x + y;
6 }
7
8 int sub(int x, int y)
9 {
10    return x - y;
11 }
12
13 void print(char *fmt, char *name, int arg1, int arg2, int res)
14 {
15    printf(fmt, name, arg1, arg2, res);
16 }
17
18 void main(void)
19 {
20    print("%s(%d, %d) = %d\n", "add", 10, 1, add(10, 1));
21    print("%s(%d, %d) = %d\n", "add", 10, 5, add(10, 5));
22    print("%s(%d, %d) = %d\n", "sub", 10, 1, sub(10, 1));
23    print("%s(%d, %d) = %d\n", "sub", 10, 5, sub(10, 5));
24 }
```

図5 簡単化前のソースプログラム
Fig. 5 Target source program.

```
1 #include <stdio.h>
2
3 int add( int y)
4 {
5     return 10 + y;
6 }
7
8 int sub( int y)
9 {
10    return 10 - y;
11 }
12
13 void print( char *name,  int arg2, int res)
14 {
15    printf("%s(%d, %d) = %d\n", name, 10, arg2, res);
16 }
17
18 void main(void)
19 {
20    print( "add",  1, add( 1));
21    print( "add",  5, add( 5));
22    print( "sub",  1, sub( 1));
23    print( "sub",  5, sub( 5));
24 }
```

図6 簡単化後のソースプログラム
Fig. 6 Simplified source program.

の処理系でコンパイルし，図5を入力して実際に得られた結果である．

6. おわりに

本論文では，ソフトウェアに対する操作を直観的かつ簡潔に記述するための言語としてソフトウェア操作言語を提案した．まず，細粒度リポジトリに基づくCASEツールプラットフォームSapidについて説明をし，ソフトウェア操作言語の位置付けを示した．次に，ソフトウェア操作言語の仕様について述べ，具体的な記述について説明をした．最後に，ソースプログラムの簡単化の例を用いて，CASEツールが容易にできることを示した．

本論文では，操作対象をソースプログラムに絞ったが，仕様書など，ソースプログラム以外の文書を扱う必要がある．ソースプログラムと異なり，仕様書などの書式は開発現場で異なるため，解析器が扱う書式を固定できない．よって，書式の定義を記述し，その記述から解析器を定義する仕組みが必要である．

本論文で示したソフトウェア操作言語では，オブジェ

クト指向技術を用いて制約を扱ったが，基本的な制約のみが扱えるだけであるため，明確に制約を記述する仕組みが必要である．制約は書式に付随するため，書式の定義の中で記述することが望ましい．また，操作の組合せによっては制約を満たさない状態になることもあり得るので，トランザクションの機構を導入することも必要である．

現在，関数仕様書の雛型の生成，program slicing，部分評価を行うCASEツールの実現に取り組んでいる．このような実用規模のCASEツールの開発に適用することで，定量的な視点からソフトウェア操作言語の有効性を示すことが，今後の課題である．

謝辞 本研究の実施にあたりEAGL事業推進機構による助成を受けた．

参考文献

1) 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，情報処理学会研究報告 ソフトウェア工学，Vol. 102, No. 7, pp. 37–42 (1995).
2) Chen, M. and Norman, R.J.: A Framework of Integrated CASE, *IEEE Software*, Vol. 9, No. 2, pp. 18–22 (1992).
3) Thomas, I. and Nejmeh, B.A.: Definition of Tool Integration for Environments, *IEEE Software*, Vol. 9, No. 2, pp. 29–35 (1992).
4) Weiser, M.: Program Slicing, *IEEE Trans. Softw. Eng.*, Vol. SE-10, No. 4, pp. 352–357 (1984).
5) Jones, N.D., Gomard, C.K. and Sestoft, P.: *Partial Evaluation and Automatic Program Generation*, Prentice Hall (1993).
6) Purtilo, J. M. and Atlee, J. M.: Module Reuse by Interface Adaptation, *Softw. Pract. Exper.*, Vol. 21, No. 6, pp. 539–556 (1991).
7) Chen, Y.F., Nishimoto, M.Y. and Ramamoorthy, C. V.: The C Information Abstraction System, *IEEE Trans. Softw. Eng.*, Vol. 16, No. 3, pp. 325–334 (1990).
8) Schwartz, J.T., Dewar, R. B. K., Dubinsky, E. and Shonberg, E.: An Introduction to SETL, *Programming with Sets*, Springer Verlag, New York (1986).
9) 重松保弘，吉見康一，吉田 将：集合指向言語SOLとその言語処理系の開発，情報処理学会論文誌，Vol. 30, No. 3, pp. 357–365 (1989).
10) Omohundro, S.M.: The Sather Language, Reference manual, Internal Computer Science Institute (1991).
11) Maeyer, B.: *Object-oriented Software Construction*, Prentice Hall (1988).

付 録

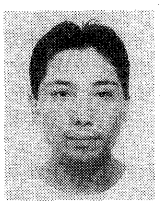
A.1 ソフトウェア操作言語構文規則

```
prog:               class_list
class_list:         class | class_list ;
                    | class_list ; class |
class:              class Ident opt_type_vars is
                      feature_list end
opt_type_vars:      (ident_list) |
ident_list:         Ident | ident_list , Ident
feature_list:       feature | feature_list ;
                    | feature_list ; feature |
feature:            type_spec | var_dec
                    | var_dec := expr | routine_dec
                    | shared_attr_dec | const_attr_dec
                    | alias alias_list
                    | private alias alias_list
                    | private var_dec
                    | private var_dec assign expr
                    | private routine_dec
                    | private shared_attr_dec
                    | private  const_attr_dec
                    | cmpnt_attr_dec(*)
                    | cmpnts_attr_dec(*)
cmpnt_attr_dec(*):  cmpnt(*) var_dec
cmpnts_attr_dec(*): cmpnts(*) var_dec
cmpnt(*):           component | relative
cmpnts(*):          components | relatives
alias_list:         alias_dec
                    | alias_list , alias_dec
alias_dec:          Ident Ident
type_spec:          Ident | $ type_spec
                    | Ident ( type_spec_list )
type_spec_list:     type_spec
                    | type_spec_list , type_spec
var_dec:            single_var_dec
                    | mult_ident_list : type_spec
single_var_dec:     Ident : type_spec
mult_ident_list:    ident_list , Ident
shared_attr_dec:    shared var_dec
                    | shared var_dec := expr
var_dec_list:       var_dec | var_dec_list ;
                    | var_dec_list ; var_dec
routine_dec:        Ident is  block  end
                    | Ident ( var_dec_list )
                      is block end
                    | single_var_dec is block end
                    | Ident ( var_dec_list )
                      : type_spec is block end
const_attr_dec:     constant var_dec := expr
block:              Ident is block end | stmt_list
                    | block except ( single_var_dec )
                      then stmt_list
stmt_list:          stmt  | local_dec | stmt_list ;
                    | stmt_list ; stmt
                    | stmt_list ; local_dec |
stmt:               Ident | assignment | conditional
                    | loop | switch | break
                    | expr break | return | call
                    | assert | debug | foreach(*)
local_dec:          var_dec | var_dec := expr
assignment:         expr  := expr
conditional:        if expr then block
                      elsif_part else_part end
elsif_part:         elsif_part elsif expr then block |
else_part:          else block |
loop:               until expr loop block end
                    | loop block end
switch:             switch expr when_part else_part end
when_part:          when_part when
                      exp_list then block |
assert:             assert ( Ident ) expr end
debug:              debug ( Ident ) block end
foreach(*):         foreach Ident : type_spec
                      in cexpr do block end
call:               Ident ( exp_list )
                    | cexpr . Ident arg_vals
                    | type_spec :: Ident arg_vals
arg_vals:           ( exp_list ) |
exp_list:           expr | exp_list , expr
expr:               cexpr | nexpr
cexpr:              Ident | Char_Const | Int_Const
                    | Real_Const | Bool_Const
                    | Str_Const | call | aref
                    | ( expr )
                    | ( Ident elt_type_spec(*)
                      in cexpr aggre_cond(*) )
elt_type_spec(*):   : type_spec |
aggre_cond(*):      '|' expr |
nexpr:              not expr | expr < expr
                    | expr > expr | expr <= expr
                    | expr >= expr | expr = expr
                    | expr /= expr
                    | expr and expr | expr or expr
                    | - expr | + expr | expr + expr
                    | expr - expr | expr * expr
                    | expr / expr
aref:               cexpr [ exp_list ] | [ exp_list ]
```

A.2 書式クラス定義一覧

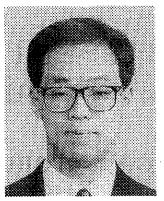
```
PROGRAM
    component files: SOURCE_FILE;
    component name: ATTRIBUTE;
SOURCE_FILE
    components blocks: BLOCK;
    components functions: FUNCTION;
    component name: ATTRIBUTE;
FUNCTION
    components arguments: DECLARATION;
    components blocks: BLOCK;
    relative a_type: A_TYPE;
    relatives callers: FUNCTION_CALL;
    component name: ATTRIBUTE;
    get_all_arguments: AGGREGATION(VARIABLE) ;
    get_an_argument(i: INT): DECLARATION;
    get_an_argument_variable(i: INT): VARIABLE;
    remove_an_argument(i: INT);
    get_all_variables: AGGREGATION(VARIABLE);
    get_all_functioncalls: AGGREGATION(FUNCTION_CALL);
    is_defined: BOOL;
    get_type_of_result: A_TYPE;
    get_declaration_str: STR;
BLOCK
    components declarations: DECLARATION;
    components blocks: BLOCK;
    components expressions: EXPRESSION;
    components labels: LABEL;
    component sort: ATTRIBUTE;
    relative composit_block: BLOCK;
    is_IF_block: BOOL;
    get_IF_condition: EXPRESSION;
    get_IF_then_block: BLOCK;
    get_IF_else_block: BLOCK;
    is_SWITCH_block: BOOL;
    get_SWITCH_condition: EXPRESSION;
    get_last_expression: EXPRESSION;
    get_a_BLOCK: BLOCK;
    is_BASIC_block: BOOL;
    is_DECLARATION_block: BOOL;
    is_FOR_block: BOOL;
    is_WHILE_block: BOOL;
    is_DO_block: BOOL;
    is_COMPOUND_block: BOOL;
EXPRESSION
    components function_calls: FUNCTION_CALL;
    components variables: VARIABLE;
    components members: MEMBER;
    components constants: CONSTANT;
    components types: A_TYPE;
    components expressions: EXPRESSION;
    components operators: OPERATOR;
    component sort: ATTRIBUTE;
    is_BREAK_expression: BOOL;
    is_RETURN_expression: BOOL;
    is_a_CONSTANT: BOOL;
    get_a_CONSTANT: CONSTANT;
DECLARATION
    components functions: FUNCTION;
    components variables: VARIABLE;
    components constants: CONSTANT;
    components types: A_TYPE;
    component a_type: A_TYPE;
A_TYPE
    component type_st: A_TYPE;
    components constants: CONSTANT;
    components members: MEMBER;
    component name: ATTRIBUTE;
    component sort: ATTRIBUTE;
    component qualifier: ATTRIBUTE;
    component a_type: ATTRIBUTE;
    component length: ATTRIBUTE;
    component sign: ATTRIBUTE;
    relative a_declaration: DECLARATION;
    relative type_use: DECLARATION;
    to_str:STR;
    components constants: CONSTANT;
    component name: ATTRIBUTE;
    component value: ATTRIBUTE;
    component a_type: ATTRIBUTE;
    is_string: BOOL;
VARIABLE
    relative a_type: A_TYPE;
    relatives expressions: EXPRESSION;
    component name: ATTRIBUTE;
    component storage_class: ATTRIBUTE;
    component scope: ATTRIBUTE;
    is_global: BOOL;
    is_local: BOOL;
    get_declaration_str: STR;
MEMBER
    relative a_type: A_TYPE;
    component name: ATTRIBUTE;
    get_declaration_str: STR;
FUNCTION_CALL
    components arguments: EXPRESSION;
    relative function: FUNCTION;
    get_an_argument(i: INT): EXPRESSION;
    remove_an_argument(i: INT);
LABEL
    component a_constant: CONSTANT;
    component name: ATTRIBUTE;
    component sort: ATTRIBUTE;
    is_NAMED_label: BOOL;
    is_CASE_label: BOOL;
    is_DEFAULT_label: BOOL;
OPERATOR
    component sort: ATTRIBUTE;
```

(平成7年3月3日受付)
(平成7年7月7日採録)



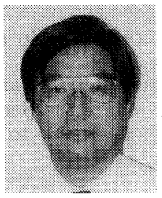
吉田　敦（学生会員）

1969年生．1991年名古屋大学工学部卒業．1993年同大学院博士前期課程修了．現在，同大学院博士後期課程在学中．ソフトウェア開発環境に関する研究に興味を持つ．



山本晋一郎（正会員）

1962年生．1987年名古屋大学工学部卒業後．同大学大学院に進学，1991年同大学助手．プログラミング言語処理系，ソフトウェアの形式的開発手法，ソフトウェア開発環境に関する研究に従事．電子情報通信学会，日本ソフトウェア科学会各会員．



阿草　清滋（正会員）

昭和22年生．昭和45年京都大学工学部電気工学第二学科卒業．昭和49年より同情報工学科に勤務．平成元年より名古屋大学教授．現在，情報工学科に勤務．工学博士．ソフトウェア工学，特に仕様化技術，再利用技術に関する研究に従事．電子情報通信学会，日本ソフトウェア科学会各会員．