ソフトウェア工学 64－25
（１９８９．２．３）

ソフトウェアの変更支援
－影響箇所の自動修正－

永良　裕[x]，天満　隆夫[x]，佐藤　康臣[x]，田中　稔[xx]，市川　忠男[xx]
[x] 広島大学大学院　[xx]広島大学工学部

ソフトウェア開発において変更にともなうプログラマの負荷を軽減するために、変更支援システムを開発した。本システムは、変更による影響の伝播を属性リレーションを用いて計算し、影響箇所の共通部分をコピールールを用いて保存することによって自動修正を行なう。

A Support for Software Modification
-Automatic Correction of Side Effects-

Yutaka NAGARA, Takao TENMA, Yasuomi SATO,
Minoru TANAKA, and Tadao ICHIKAWA

Faculty of Engineering, Hiroshima University
Shitami, Saijyo-cho, Higashi-Hiroshima, 724, Japan

In order to reduce the programmer's burden caused by the modification of programs, we have developed a software modification support system on the basis of the integrated programming environment named MUSE. This system detects and corrects all side effects caused by the modification automatically. Attribute-relations and copy rules are used for the detection of side effects and correction of errors, respectively.

1．はじめに

ソフトウェアの需要が増大するにつれて、ソフトウェア開発の生産性、信頼性の向上が求められている。近年、ソフトウェア開発の生産性の向上を目的として、特定言語向きの機能を提供する統合化プログラミング環境が開発されている。代表的な統合化プログラミング環境として、Ｉｎｔｅｒｌｉｓｐ－Ｄ[1]やＳｍａｌｌｔａｌｋ－８０[2]などがある。しかし、統合化環境は特定の言語ごとに構築しなければならないという問題がある。そこで、特定の言語に関する記述を与えることによって、その言語向きの環境を生成し得るようなメタプログラミング環境についても研究されている。その中で特にＳｙｎｔｈｅｓｉｚｅｒ－Ｇｅｎｅｒａｔｏｒ[3]やＧａｎｄａｌｆ[4]などがよく知られている。

我々は、複数のソフトウェア記述の支援、インクリメンタル性、統合性、言語依存性を目的として、メタプログラミング環境ＭＵＳＥを開発している[5][6]。現在、ＭＵＳＥは言語指向エディタを中心に構築されており、コーディングフェーズにおける編集、実行のための機能を提供している。今回、ソフトウェアの変更時におけるプログラマの負荷を軽減するために、ＭＵＳＥ上に変更支援システムを実現した。

ソフトウェアに対してプログラマが変更を加えると、変更された部分とその他の部分の間に不整合が生じることがあり、生じた不整合を修正する作業量はソフトウェアの大きさに比例して大きくなる傾向があり、それらの作業はプログラマにとってまったく非創造的な作業である。

ソフトウェアの変更時の負荷を軽減する方法としては、これまでにモジュール化の徹底やオブジェクト指向プログラミングが考えられているが、今だ十分であるとはいえない。以上のような理由から、ソフトウェアの変更に対して支援を行なうことにより、プログラマの生産性が飛躍的に向上すると考えられる。ところが、変更支援に対する研究例は、ＴＭＭ（ＤＲＡＣＯプロジェクト）[7]やＫＢＥｍａｃｓ（プログラマーズアプレンティスプロジェクト）[8]などに見られるだけでまだ十分とはいえない。

我々が提案する変更支援システムは、プログラマが行なった変更によるソフトウェア内の複数の影響箇所を発見し、それらを自動修正することにより、プログラマの負荷を軽減し、プログラマが本質的に重要な作業に専念できるようにすることを目的とする。本システムの利用によりソフトウェアの生産性と信頼性が向上すると考えられる。

変更支援機能の実現のために、本システムは、ソフトウェア部品単位での編集にもとづき、属性リレーションによってソフトウェア内の属性伝播と制約のチェックを行ない、変更前の部品と変更後の部品の間の共通部分をコピールールを用いて保存することによって自動修正を行なうといった方法をとっている。

以下、本稿ではまずソフトウェアの変更と変更時の支援について考察し、次に変更支援システムについて述べる。次にソフトウェア部品について説明し、つづいて実際のシステムの動作について説明を行ない、最後にまとめを行なう。

2．ソフトウェアの変更とその支援

ソフトウェアの変更とは、プログラマがある意図によってソフトウェアに対して編集を加えることである。ここでは、変更とそれに対する支援機能について考察する。

2．1　ソフトウェアの変更

本節では、ソフトウェアに対して加えられる変更に関して一般的な議論を行なう。

まず、ソフトウェアの変更に最も関連が深いのはソフトウェアの保守である。保守とは、ソフトウェアの運用段階において、なんらかの形でプログラムに対して変更を加えることである。保守に関して、ソフトウェアの設計、コーディング、デバッグなどのコストに対して、保守の占めるコストが非常に大きいと言われているが、これはソフトウェアの変更が難しいために、保守ににかかるコストが大きくなっていると考えられる。また保守だけでなく、コーディングやデバッグにおいても様々な理由でプログラムに変更が加えられる。この場合にも、変更の影響によりバグが生じ、プログラマにとって大きな負荷になっている。このように、ソフトウェアのライフサイクル全般にわたり変更の難しさが問題となっている。

ソフトウェアに対し様々な変更が加えられるが、その主だった要因は以下のようなものである。

（１）効率の改善
（２）移植
（３）機能の変更
（４）バグの修正
（５）その他

また、変更を分類すると、ソフトウェア全体の機能が変更前と変更後で等価な変更と、ソフトウェア全体の機能が変るものとに分けられる。等価な変更の例としては、ソートの実現アルゴリズムをバブルソートからクイックソートに変更する場合やあるオブジェクトを表現するデータ構造を効率のよい等価なデータ構造へ変更する場合などが考えられる。非等価な変更は、ソフトウェアの機能が追加、削除されることなどで、その一例を挙げると、オブジェクト指向パッケージにおいて単一継承を多重継承に変更する場合がある。一方、プログラムがデータ構造とアルゴリズムから成るといった点から見ると、変更をアルゴリズムの変更とデータ構造の変更とに分類することが可能であろう。

以上、変更の原因とその大まかな分類について述べた。しかし、ソフトウェアの変更に関して次のような問題が存在する。

（１）ひとつの変更がプログラム内の多くの場所に影響を与える。例えば、ある変数の型宣言を変更すると、変更された変数を参照しているプログラム内の全ての部分に影響が伝播し、場合によってはその部分を修正しなければならない。
（２）現在、変更はプログラムに付属するドキュメントによって行なっているが、それらは機械的処理ができない。例えば、モジュール間の参照関係などがドキュメントとして管理されている場合でも、これらのドキュメントに記述された内容をツールが取り扱うことができないので、変更時にはドキュメントの解釈を人手に頼らなければならなくなる。
（３）設計時の決定事項などが直接プログラム上に反映されていないため、プログラマは変更時に設計時の決定事項を調べる必要があり、変更時のプログラマの負荷が大きくなっている。
（４）何度も変更されたプログラムは構造が不明確になる。モジュール化が行なわれているソフトウェアであっても、変更されることによりモジュール化の利点が失われることがある。したがって変更が度重

なると、それ以後の変更がさらに困難になる。

２．２　変更に対する支援機能

変更支援に要求される機能として、次のような機能が考えられる。

ソフトウェアの一部が変更された時に、
(１)変更が影響する箇所を表示する機能
(２)不整合に対する修正案を提示する機能
(３)不整合を自動修正する機能
(４)変更の履歴を表示する機能
などがある。本システムでは(１)ならびに(３)の機能を支援する。

支援を行なう対象としては、プログラムコード、設計など様々なソフトウェア記述の一つの記述に対する支援、設計の変更をプログラムコードに反映させるといった異種の記述間に対する支援とが考えられる。本システムでは、コーディングフェーズにおけるプログラムコードのみを支援対象とする。また、主に支援対象となる変更の種類は、２．１で分類した変更の中のデータ構造を中心とする変更である。また、アルゴリズム変更で機能の非等価な変更については、完全な自動修正は不可能であるが、支援がある程度可能である。

３．変更支援システム

ここでは、ＭＵＳＥ上に実現されている変更支援システムについて述べる。

３．１　ソフトウェア表現

ここでは、ＭＵＳＥで取扱うソフトウェアがどのように表されるかについて説明する。ＭＵＳＥは一つのソフトウェアに対して、内部表現と外部表現という二つの表現形式をもっている。あるソフトウェアの一部の内部表現と対応する外部表現を図１に示す。

内部表現とは、ＭＵＳＥ内部のソフトウェア表現である。システム内部では、ソフトウェアに対して各ツールが種々の操作を行ないやすいように、ソフトウェアは属性付き抽象構文木をベースとし、任意のノード間を結合するリンクを付加したネットワーク構造で表現されている。抽象構文木の各ノードは、後述する部品に関する記述である部品クラスのインスタンスであり、一つの構文要素や部品などを表す。ノードは、クラス名を表すラベル、ノード間を結合し木構造を表す構造リンク、任意のノード間を結合する意味リンク、型や名前などの属性などの情報を保持している。

外部表現は、ユーザインタフェースを通してソフトウェアをプログラマに表示する際の表現である。プログラマに対しては、ソフトウェアはテキストあるいはデータフローグラフといった人間にとってわかりやすい形で表示される。これは、内部表現をフォーマット規則にしたがって変換することによって生成される。

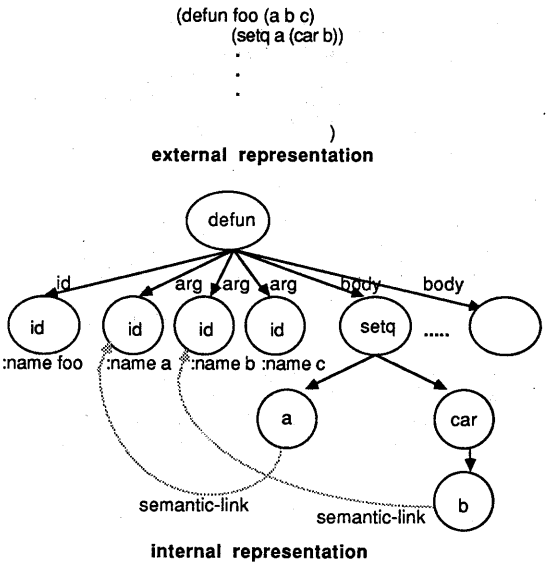


Fig.1 Software Representation

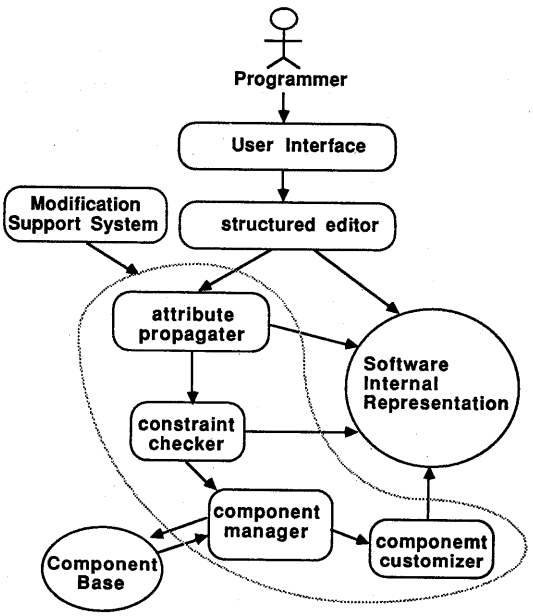


Fig.2 System Organization

３．２　システム構成

変更支援システムのシステム構成を図２に示す。プログラマがエディタを用いてプログラムに編集を加えると、エディタのよって破線で囲まれた変更支援システムが起動される。変更支援システムは、以下の要素から構成されている。

１)属性伝播ツール

エディタから直接起動される。変更されたノードを受け取り、そのノードから部品記述の伝播式に基

づいて内部表現内で属性を伝播させる。
2)制約チェッカ
属性伝播ツールから起動される。ソフトウェア内の制約を評価し、伝播された属性と制約の間のチェックを行なう。もし違反が生じている場合には、プログラマに対して意図的な変更であるかどうかを質問し、意図的な変更であれば部品検索ツールを呼び出す。意図的な変更でなければ、エラーメッセージをプログラマに返し、行なわれた変更をキャンセルする。
3)部品管理ツール(部品検索ツール)
属性が制約を満足しない場合に制約チェッカから起動される。制約違反の生じたノードの自動修正のために、部品ベースから制約を満足させる部品の検索を行なう。制約を満足するものが複数存在する場合には、プログラマに選択させる。
4)部品適用ツール(自動修正ツール)
自動修正を行なう際に、部品管理ツールによって検索された部品と制約違反の生じた変更前の部品との間の共通部分を変更前の部品から検索された部品へ部品記述中のコピールールにしたがってコピーする。もしコピールールが存在しない場合には、プログラマに対してノードの展開を促す。

4．ソフトウェア部品

今まで、ＭＵＳＥでは構文要素単位での編集機能を提供していた。変更支援機能を実装するにあたり、アルゴリズムやデータ構造の変更などのように構文要素よりも大きな単位での変更がなされる場合が多いので、部品単位での編集が行なえるようにソフトウェア部品(以下、部品と略す)の考えを導入した。また、ソフトウェアの再利用の点でも、ソフトウェア部品の利用は有効であり、各種の研究がなされている[9]。以上により、ＭＵＳＥに特定言語に対する構文要素を含むソフトウェア部品に関する記述を与えることにより、特定の言語に対する変更支援機能をもつ環境を提供することが可能になる。
ソフトウェアを部品単位で取り扱うことによって、構文単位と意味リンクでは限界のあるソフトウェア内の様々な関係を明示的に取り扱うことが可能である。

4．1　部品の定義

部品とは、目標プログラムの一部として利用され、原型のままかあるいは形をかえて目標プログラムの一部となるものである[10]。本研究では、ソースコードを部品化したものを取り扱う。

4．2　部品の種類と分類

本システムでは、特定の言語に対する部品記述の集合を部品モジュールと呼ぶ。部品の数が増加し部品モジュールが膨大になると、その中の部品を検索するためには、部品を効果的に分類する必要がある。また、その分類法は、プログラマに対してわかりやすいものでなければならない。本研究では、softech社の部品ライブラリシステムで用いられているファセット分類の方法を用いている[11]。この分類法は図書館などで一般に用いられており、分類対象の集合が大きく、絶えず変化しているものに向いている。ファセット分類法とは、ファセットとよばれる対象に関する断片的な記述をあらかじめ定義してある引用順序にしたがって並べるものである。プログラマは、ファセット記述をシステムに入力することによって、必要な部品を検索することができる。そのためには、ファセット表と引用順序の定義が必要になる。本研究では、以下のような引用順序とファセットとその値を定義した。
引用順序
　(部品型　機能　対象　媒体)
部品型ファセット
　関数　制御パターン　データ構造　構文要素
機能ファセット
　挿入　削除　展開　生成　交換.....
対象ファセット
　リスト　レコード　整数　実数.....
媒体ファセット
　二分木　リスト　バッファ.....
例えば、リストに整数を挿入する関数の部品を検索する場合には
　(関数　挿入　整数　リスト)
というファセットをシステムに与えればよい。また、各ファセットにはワイルドカードの意味をもつ＊を用意してある。

4．3　部品記述

ソフトウェア部品を定義する部品記述について述べる。これはまた、制約チェック、構文チェック、変換フォーマット、自動修正のための情報などをもつ。部品記述とソフトウェア表現のノードとの関係は、オブジェクト指向におけるクラスとインスタンスの関係に相当する。連想リストの生成関数make-a-listの部品記述の例を図３に示す。部品に関する記述は以下の記述からなる。

部品名
　部品の名前を表わす。
スーパークラス
　スーパークラスを表し、下位クラスは上位クラスの属性、メソッドなどの情報を継承する。
ファセット
　部品の分類に用いられ、部品検索の際のキーとなる。
構造リンク
　部品の可変部分を表わし、リンク名および接続可能なクラス，接続可能なノードの個数を記述する。これらの制約をリンク制約と呼ぶ。図では、consリンクの先にconsクラスのサブクラスが接続されなければならず、その個数は０個以上であることを表している。

```
class      make-a-list

:super-class   (type-generate-func)
:faset   (fun gen a-list*)
:struct   ((cons :class cons :range (0  infi)))
:semantic   ((call-def) (return-type))
:copy-rule   (:index ((node cons car) 'name)
                     :value ((node cons cdr) 'type))
:attribute   ((return-type))
:constraint   (:link ((cons car)  (cons cdr)  self)
                     :prop ($x  $y  a-list)
                     :const (symbol  t  a-list))
:method   ((code-gen (lambda ()..)))
:probe   nil
:format  .....
```

Fig.3　Class description of make-a-list

意味リンク
関数の呼び出し部分とその定義部分といった意味的な関係を記述する。図では、call-defとreturn-typeの二つのリンクがあることを表している。
コピールール
クラス置換の時の構造リンクの自動穴埋めに使われる異なるクラス間の対応を記述する。５章の変更支援で詳しく述べる。
属性
変数の型や関数のリターンタイプなどの属性を記述する。
制約
自分の属性と他の部品の属性間の満たさなければならない関係を属性リレーションを用いて記述する。５章の属性伝播で詳しく述べる。
メソッド
クラスのインスタンスであるノードに対してメッセージが送られたときに実行される手続きをラムダ式で記述する。
プローブ
一種のデモンであり、リンク先のノードが変更されたときに自動的に起動される手続きを記述する。
フォーマット情報
プリティプリンタが内部表現を外部表現に変換するときに参照する変換規則である。

４．４　部品ベース

部品ベースは、部品クラス内の上位クラスの関係にしたがって階層化されている。この階層化により、部品クラス検索の効率化、継承により部品クラスの記述量を減少させることができるといった利点がある。また変更支援においても、上位クラスは下位クラスの共通の性質をもっているので、変更の自動修正時の共通部分のコピーにも有効である。

上位クラスは、構文要素であればＯＲオペレータからなる構文規則から導かれ、関数型の部品と制御パターン型の部品であればシステムに与えられる機能ファセットの包含関係によって決定される。データ構造型の部品は、言語ごとに与えられる型の階層にしたがって決定される。

５．変更支援システムの動作

ここでは、実際のプログラム開発の一部を例に、変更支援システムの動作を説明する。

５．１　シナリオ

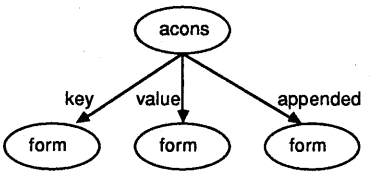
例としてして取り上げるのは、オブジェクト指向の機能を実現するためのパッケージであり、クラス登録、オブジェクト生成、メッセージを実現する関数を作成するものである。５．３と５．４の修正支援(A),(B)ではデータ構造の変更に関する支援を、５．５の修正支援(C)では機能の変更の例を説明する。

５．２　属性伝播

変更による影響点を計算するために、抽象構文木内の変更されたノードの属性を伝播する。本システムでは、以下に示すような属性リレーションを用いた方法を利用している。なお、属性の評価と伝播は編集操作ごとに行なわれる。図４にLisp関数aconsの属性リレーションとその内部表現を示す。この属性リレーションはノードの属性に対する制約と、接続されるノード間の属性の伝播の規則を表す伝播式とにより構成される。関数aconsは、自身のノード

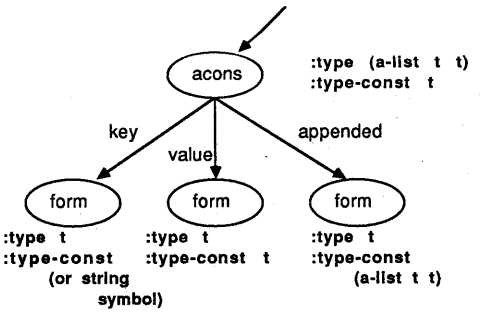
link name	self	key	value	appended
propagation	(a-list $x $y)	$x	$y	(a-list $x $y)
constraint	(a-list t t)	(or symbol string)	t	(a-list t t)

(a) attribute-relation of acons

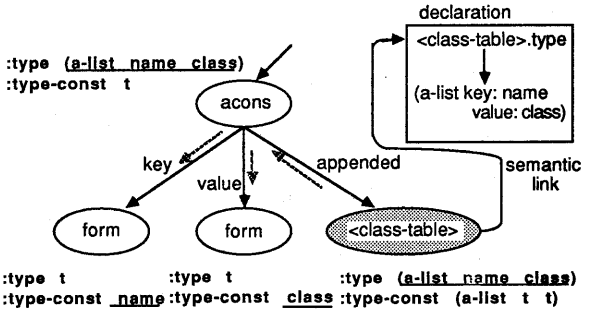


(b) internal representation of acons

Fig.4 Attribute-Relation



(a) initial constraint of acons



(b) attribute propagation

Fig.5 Attribute Propagation

の型属性はa-listのサブタイプでなければならない、keyリンクの先のノードの属性はsymbolまたはstringのサブタイプでなければならない、valueリンクの先のノードの型属性はt型のサブタイプすなわち何でもよい、appendedリンクの先のノードの型属性はa-listのサブタイプでなければならない、という制約をもっていることを表している。

また、伝播式は＄で始まる変数$xと$yを用いることにより、a-listのcar部分の型属性とkeyリンクの先の型属性が同じであり、cdr部分の型属性がvalueリンクの先の型属性と同じであり、appendedリンクの先の型属性がacons自身の型属性と同じになるように属性が伝播することを表している。

以上の属性リレーションの記述をもとに、属性伝播は次のように行なわれる。図５(ａ)のように関数aconsが展開されると、リレーション中の制約の記述を参照して、各構造リンクの先のノードのtype-const属性にaconsの制約を伝播する。type-const属性はノードに他のノードからかかっている制約を保持している。ノードが置き換えられたとき、新しいノードのtype属性がこのtype-const属性を満足するか否かの型制約のチェックに用いられる。

次に図５(ｂ)のようにappendedリンクの先のノードが変数<class-table>に置き換えられると、<class-table>のtype属性がtype-const属性を満足しているか否かのチェックが行なわれる。この例では満足しており、ノードが置き換えられる。ノードの型属性が変更されるので、型属性にセットされているプローブが起動される。プローブはaconsの伝播式の記述を参照し、(a-list $x $y)と(a-list name class)でパターンマッチを行ない、$xにname、$yにclassを代入し、keyリンクの先のtype-const属性にname、valueリンクの先のtype-const属性にclassを<class-table>の制約として伝播させる。また、acons自身には親ノードへの属性伝播を行なうためにtype属性に伝播させる。このとき、伝播されるtype属性はaconsのノードのtype-const属性を満足するかのチェックが行なわれる。この例では満足しているのでtype属性に伝播させる。型制約のチェックは、type属性またはtype-const属性が変更されるときに行なわれる。言語ごとに定義される型の階層関係を参照して、type属性がtype-const属性のサブタイプになっていれば型制約を満足していると考える。以上の処理の後、aconsのtype属性が変更されるので、さらにプローブが起動され再帰的に属性が伝播する。

５．３　修正支援(Ａ)

ここでは、オブジェクト指向パッケージで用いられるclassのデータ構造がa-listからstructに変更される場合のシステム内部の動作について説明する。

プログラマが図６(ａ)のようにclassの型宣言をdefnewtypeからdefstructに変更すると、図６(ｂ)の構文木上で同じリンク名を持つノードをコピーする。defnewtypeのidリンクの先はidクラスのノードであり、defstructのidリンクの先はidクラスであるというリンク制約を満たしているので、defnewtypeのidリンク先のノードがdefstructのidリンク先にそのままコピーされる。しかし、defineリンクの先のa-list-defはリンク制約に違反しており、defstructのリンク制約にしたがってstruct-defクラスに置換される。

図７に型定義クラスの階層の一部を示す。a-list-defのコピールールは、a-list-defのノードのconsリンクの先のノードのcarリンクの先のノードのname属性が、a-list-defのスーパークラスであるindexed-setのindexに相当することを表している。このコピールールを適用することにより、図８(ａ)の構文木は、図８(ｂ)の中間の木構造に変換される。さらにstruct-defのコピールールは、struct-defのノードのfieldリンクの先のノードのname属性がindexに相当することを表しているので、このルールを適用することにより図８(ｃ)のような木構造に変形される。以上の手順で両方のクラスの共通する部分を構文木上でコピーすることによって宣言部の自動修正を行なう。

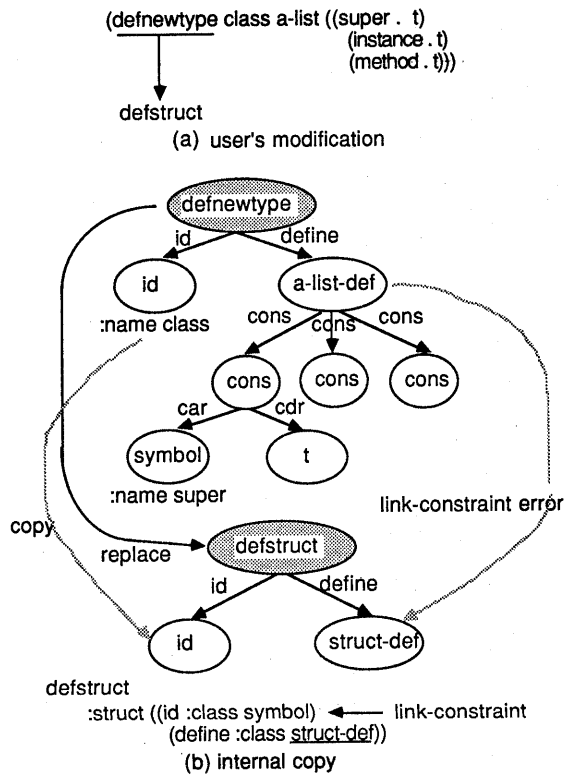


Fig.6 Support A

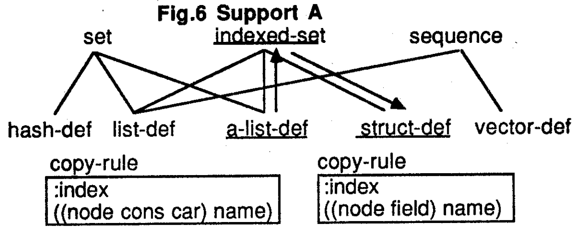


Fig.7 Type-def Class Hierarchy

５．４　修正支援(Ｂ)

宣言部が変更されると、５．２で述べた属性伝播によって、意味リンクを通じてclass型の変数を使用している部分やclass型を返す関数に属性が伝播する。例えば、データ構造の生成関数であるmake-a-listに型属性が伝播する。make-a-listは自分自身に対する型制約としてa-listを持っているので、伝播した型structを満足しない。そこで、クラス階層

にしたがって次の手順で置き換えるべきクラスを検索する。

ＳＴＥＰ１
　置換前のクラス(a-list)のスーパークラスの型属性が伝播してきた型制約のスーパータイプであるかの言語ごとに定義される型階層を参照してチェックを行なう。
満たせばＳＴＥＰ２へ
満たさないならば、さらにスーパークラスについてＳＴＥＰ１を行なう。

ＳＴＥＰ２
　ＳＴＥＰ１で発見されたクラスの型属性は伝播した型制約のスーパータイプであるのでその下位クラスにより詳細に制約を満たすクラスがある可能性があるので、その下位クラスに幅優先探索を行なう。より詳細にとは、型階層において型制約と型属性の距離が近いということである。

以上の検索によってクラスmake-classが見つけられ、上記と同様な手順でコピーが行なわれ、自動修正される。

５．５　修正支援(C)
　ここで例として取り上げるのは、オブジェクト指向パッケージにおいて、単一継承を多重継承に変更する場合である。これは、(A)と(B)で取り上げたデータ構造の変更を中心とした等価な変更ではなく、非等価な機能拡張である。この変更支援が、どのように行なわれるか説明する。
　まずプログラマは、図９の線形探索を実現する制御パターンの部品linear-searchをネットワーク探索の部品dfsで置き換える。ここで、プログラマは部品を検索する際に、ファセットとして(制御パターン　探索　ネットワーク　＊)をシステムに入力する。システムは部品階層中を検索し、ネットワーク探索用の部品であるdfsとbfsをプログラマに提示する。
　一般に線形探索はネットワーク探索の特殊形と考えられるので、図９に示すように両部品は共通する部分を持っている。ここでは、searchクラスの探索

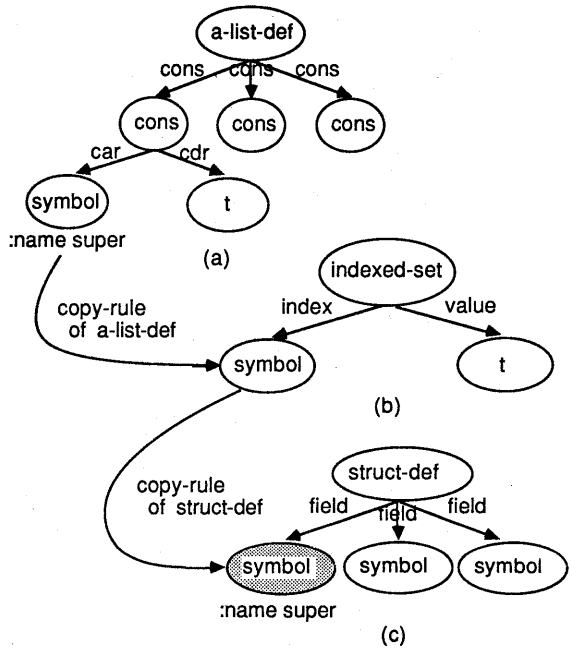


Fig.8 Copy　Mechanism

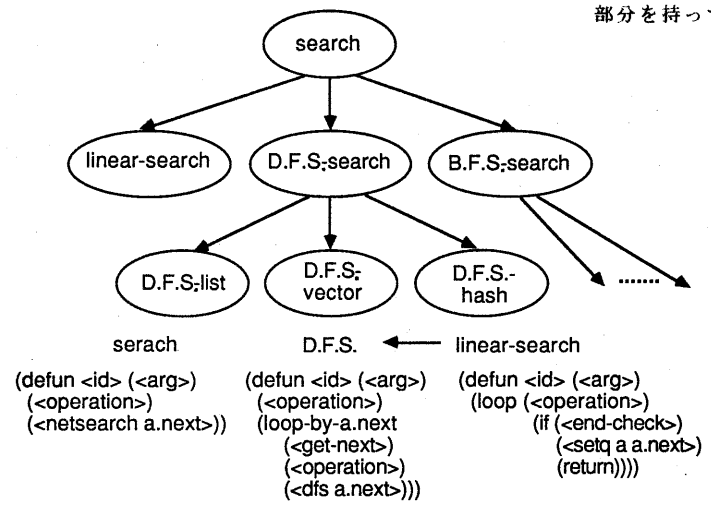


Fig.9 Search Component

のキー(a a.next)と探索時に行なう操作(operation)が両部品に共通している。これらを変更時にコピーすることによって自動修正を行なう。しかし、次に探索すべきノード集合のデータ構造などが決定されていないので、図１０のようにdfsに置き換えた場合loop-by-nextの先のノードが決定されない。このような場合、システムが提示する制約を満足させるクラス候補の中からプログラマが一つを選択しなければならない。ここでは、ノード集合がリストになっているので、プログラマはdolistを選択する。

６．おわりに

以上、本稿で述べたシステムは、ソフトウェアに対する変更を以下のアプローチによって、効率的に自動修正を行なっている。

１)ソフトウェア部品と部品ベースの利用
２)属性伝播と制約のチェック
３)コピールールにもとづく木構造の変形を行なうことにより部品間の共通部分の保存し、自動修正を行なう。

現在、変更支援のシステムをＳＵＮ－３、ＵＮＩＸ上にＫＣＬとＣ言語を用いて実現中である。部品ベースは学生実験程度のプログラムが作成可能な程度のものを構築している。

今後の課題として、機能拡張などの非等価な変更についての支援がある。また、本システムは部品ベースに依存する部分が多いため、充分な部品ベースを構築する必要がある。この支援として、部品の登録について考察しなければならないと考えている。今後、実際のソフトウェア開発に適用しシステムの評価を行なう必要がある。

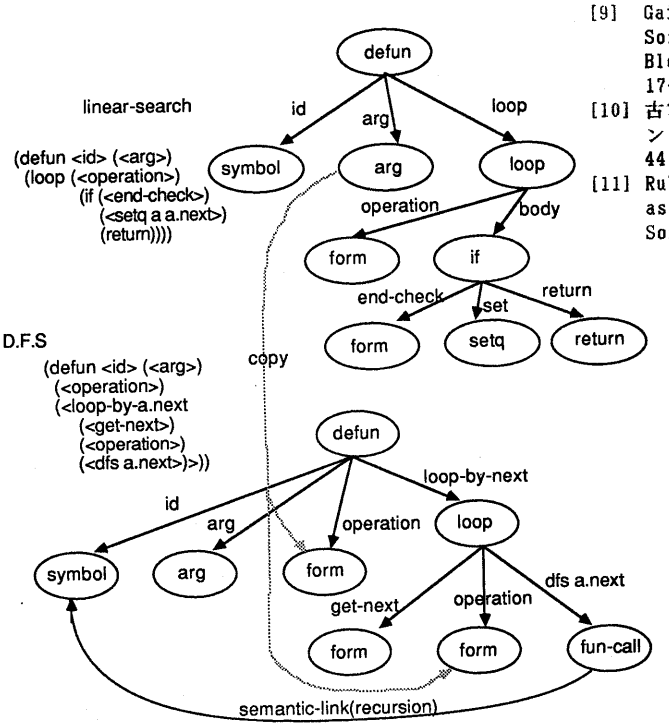


Fig.10 Support C

参考文献

[1] W.Teitelman and L.Masinter, "The Interlisp Programing Environment," Computer, vol. 14, no.4, pp25-34, 1981.

[2] A. Goldberg and D. Robinson, "Smalltalk-80: The Language and Its Implementation," Addison-Wesley, Reading, Mass. 1983.

[3] T. Reps and T. Teitelbaum, "The Synthesizer Generator," Proc., ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Devlop. Env., pp42-48, 1984.

[4] A. N. Habermann and D. Notkin "Gandalf:Software Development Environments," IEEE Trans., Software Eng., vol.SE-12, no.12 pp. 1117-1125 1986.

[5] Takao Tenma, Hideaki Tsubotani, Minoru Tanaka and Tadao Ichikawa, "A System for Generating Language-Oriented Editors," IEEE Trans. Software Eng., vol.SE-14, no.8, 1988.

[6] 佐藤，天満，永良，坪谷，田中、市川，"統合化プログラミング環境ＭＵＳＥの開発" 情報処理学会ソフトウェア工学研究会資料，58-19，昭63-02.

[7] Guillermo Arango, Ira Baxter, Peter Freeman, and Christopher Pidgeon,"TMM:Software Maintenance by Transformation," IEEE Software, vol.3, no.3 pp.27-38, 1986.

[8] R. C. Waters, "The Programmer's Apprentice: A Session with KBEmacs," IEEE Trans. Software Eng., vol.SE-11, no.11, pp.1296-1320, 1985.

[9] Gail Kaiser, and David Garlan, "Melding Software Systems from Reusable Building Blocks," IEEE Software, vol.4, no.4, pp. 17-24, 1987.

[10] 古宮，原田,"部品合成による自動プログラミング" 情報処理，vol.28, no.10, pp.1329-1344, 1987.

[11] Ruben Prieto-Diaz, and Peter Freeman, "Classifying Software for Reusability," IEEE Software, vol.4, no.1, pp.6-16, 1987.