仕様変更のプログラムへの写像

——仕様変更プロセスを利用したプログラム合成——

松浦 佐江子† 本位田 真一†

仕様変更の要求をプログラムに容易にかつ正確に反映させるために，仕様変更要求を設計モデル上で形式的に記述し，プログラムを合成するプロセスを再利用して，変更要求を満たすプログラムを開発する方法を広範囲言語Extended ML の枠組で提案する．Extended ML は，関数型言語Standard ML の拡張言語であり，一階等式論理による公理を定義できる．仕様変更プロセスをつぎの2つのプロセスから構成されると考える．第一は，Extended ML により定義された設計仕様を仕様変更要求を満たすように，変更対象のモジュール間の関係を使って加工するプロセスであり，これを仕様の差分定義プロセスと呼ぶ．第二のプロセスは，設計仕様からプログラムを合成する過程であり，これを合成プロセスと呼ぶ．合成プロセスは，プログラムの実行順序や制御の決定を含むプログラム作成の具体的かつ有効な事例であるので，これを再利用してプログラムを合成する．この時，事例を適切に再利用するために，仕様の差分定義プロセスを利用する．本稿では，これらのプロセスをML によるモジュール操作関数とプロセス操作関数によって定義し，仕様変更要求のプログラムへの写像を実現する．

Mapping Specification Change Requirement to Program

——Program Synthesis using Specification Change Process——

SAEKO MATSUURA † and SHINICHI HONIDEN †

Our goal is to formalize the program development method so that it may be incorporated into automatic and interactive programming environment. In this environment, a program can be altered to meet the specification change using a derivational process from the specification to the program. In this paper, a wide spectrum language Extended ML defines a single unified framework where specification, process and program can be expressed without ambiguities. In this framework, we first propose module relationships that become the ground for systematic reusing past programming experience. Module relationship is a process where a new module meeting the specification change is created from some well-known module. Because the well-known derivational process is a good example for synthesizing a program similar to the existence one, we second propose a procedure for reusing it using the above module relationships. Furthermore, alteration of process preserves the correctness of synthesizing for the changed specifications. These two processes define a mapping from the specification change requirement to a program using some module manipulation functions and process manipulation functions expressed in ML.

1. はじめに

1.1 動機と目的

われわれは仕様変更要求を満たすように，プログラムを容易にしかも正しく変更する方法の確立を目指している．ソフトウェアの開発を要求仕様から設計仕様，そしてプログラムへの段階的開発と考えると，要求仕様とプログラムの間には，つぎのような開発者のソフトウェア作成意図が含まれている．

第一は，要求仕様を開発者の考える設計モデルへ写像し，設計仕様として定義することである．

第二は，設計仕様を満たすように，プログラムを作成することである．

設計仕様を満たすプログラムの実行順序や制御は，設計仕様からプログラムを作成する過程において，開発者が決定する．そこで，仕様変更要求が既存の設計仕様からの差異として形式的に定義できたとしても，その情報だけではプログラムを機械的に変更することはできない．また，変更された仕様から新たな方法で作成されたプログラムが，はじめに作成されたプログラムに対する変更要求を満たしているとは限らない．

† 新ソフトウェア構造化モデル研究本部情報処理振興事業協会（IPA）
Information-technology Promotion Agency (IPA)

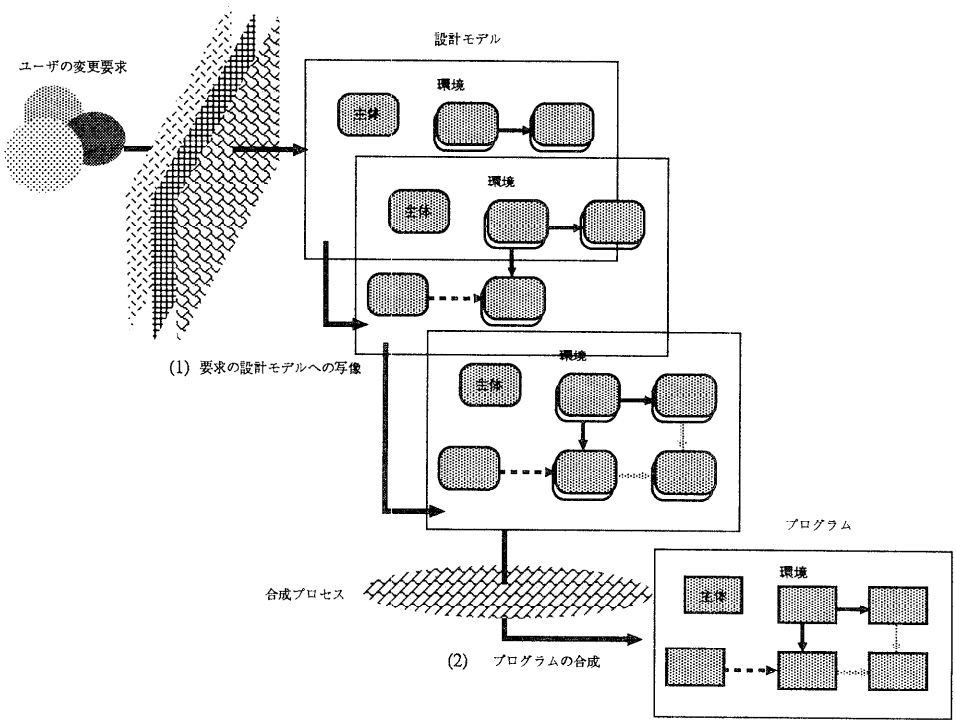

図1 仕様変更プロセス
Fig. 1 Specification change process.

これは，第二の開発者のソフトウェア作成意図を保存し，再利用していないからであり，仕様変更要求を正しくプログラムに反映したことにはならない．

そこで，仕様変更要求をプログラムに正確に反映させるためには，上記2つの開発者のソフトウェア作成意図を定義し，操作し，利用できることが必要である．

われわれはソフトウェアの仕様変更プロセスを，ソフトウェア開発における開発者のソフトウェア作成意図を記録し，操作し，利用するプロセスと捉え，広範囲言語 Extended ML[17]の枠組においてこれを定義する．そこで，仕様変更プロセスは**図1**に示すつぎの2つのステップから構成される．

（1） 要求の設計モデルへの写像

（2） プログラムの合成

（1）要求の設計モデルへの写像とは，ユーザの変更要求を開発者の視点から要求分析し，変更項目を Extended ML の設計モデル上における既存の仕様との差異として形式的に定義するプロセスである．このプロセスを**仕様の差分定義プロセス**と呼ぶ．これが，開発者の第一の意図を表している．

（2）プログラムの合成とは，（1）のプロセスによって作成された一階等式論理に基づく Extended ML の形式的仕様を，その推論規則を用いて変換し，実行可能な Standard ML*[13),15)]のプログラムを導出するプロセスである．

仕様からプログラムを合成する手順は，証明の手続きを必要とするために，簡単に自動化することはできない．しかし，手続きを一階等式論理にもとづく証明手続きとして形式的に記録することはできる．この導出の履歴を**合成プロセス**と呼ぶ．合成プロセスが，開発者の第二の意図を表している．

このような形式的開発を用いてプログラムの作成を行うことにより，導出されたプログラムが仕様を満たしていることが保証される．本稿では，第二の開発者のソフトウェア作成意図を仕様変更プロセスにおいて保持するために，既存の合成プロセスを再利用する方法を提案する．合成プロセスを再利用することによって，仕様から直接プログラムを合成するよりも容易な手続きで，かつ元のプログラムの合成意図を保存しながら変更要求を満たすプログラムを合成することができる．

* 以下，単に ML と記す．

1.2 再利用の問題点

既存の合成プロセスは，設計仕様を満たすプログラムの作成方法の成功事例である．合成プロセスを再利用するためには，事例ベース推論[10]における問題点と同様に，つぎの2つの問題を解決しなければならない．

（I） 新しい要求（＝設計仕様）に適合する事例（＝合成プロセス）を，多くの既知の事例の中からどのようにしてみつけるか．

（II） どのようにして事例を適切に修正し，新しい要求に適用させるか．

プログラム作成において，過去の作成経験を再利用することは，人工知能やソフトウェア工学の分野で，数多く提案されている．仕様・プロセスの類似性に着目した研究が，類推によるプログラミング[5]や誘導類推[4),14]であり，プログラム変換技術[16]や定理証明技術[7]に着目した研究が，設計の再利用[9]であると考えられる．これらは，上記問題点（II）に対する研究である．

一般に類推の研究は，既知の知識の一部を未知（目的）の知識に写像することによって定式化される．類推によるプログラミングでは，仕様の差異を記述された仕様上のラベルの変換として定義し，これを既知の情報として未知の情報であるプログラムを変更している．変換されたプログラムがその仕様を満たさないと，一般的なプログラムの導出方法を当てはめながら手探りでプログラムを変更している．

設計の再利用では，プログラム変換技術に基づく設計の戦略を定めて，戦略を適用しながら設計を行い，その実行履歴を蓄積し，新たな問題解決に利用している．誘導類推でも，既知の問題解決プロセスを新しい問題に適用して新たな問題解決を行っている．しかし，どちらの場合も，設計の戦略や，問題解決の判断過程といった問題に依存しない一般的な変更法則としてプロセス適用することを試みている．そのために，再利用の根拠が一般的で希薄であり，適用手順がアドホックになっている．このように系統的に再利用が行えない原因の1つは，仕様・プロセス・プログラムを統一的な枠組で扱っていないからであると考える．

上記2つの再利用の問題点は，独立した問題ではなく，互いに強く関連しているので統一的に扱わなければならない．すなわち，第一に記述言語が仕様を適切に表現でき，意味を保存したプログラムを作成できる論理的基礎をもち，さらにその作成手順が形式的に定義できることが必要である．この作成手順がプログラムと同じ言語によって表現できることで，仕様とプロセスの関係が曖昧性なく定義でき，問題（I）（II）を議論する基盤を与えることができる．

このような意味で，仕様・プロセス・プログラムを同じ言語の枠組で記述し，類似のプログラムの正しい導出を行う研究[11]もある．この研究では，仕様の一般化によって仕様の共通の性質を求め，仕様の類似性を定義しているが，仕様変更において，どのような一般化が可能であり，かつ適切であるかが明確ではない．すなわち，ここでも再利用の根拠が希薄であり，系統的に再利用が行えないもう1つの原因になっている．

（II）に対する解を与えるためには，この再利用の根拠を上述の統一基盤の上で定義し，プロセスの変更に利用できることが必要である．本稿の目的は，以下の方針によって，（II）の解である仕様変更要求を正確に反映したプログラムの生成方式を提案することである．

- 設計仕様を変更要求を満たすように加工する手順を分類し，仕様の差分定義プロセスとして定義する．
- 合成プロセスという具体的かつ有効な事例を仕様の差分定義プロセスを使って再利用する．

本稿の構成はつぎのとおりである．続いて，本論に入る前に，本稿での形式化の基礎となる言語Extended MLとMLについて説明する．2章では，要求の設計モデルへの写像を，関数型言語MLで記述されたモジュール操作によって定義する．3章では，プログラム合成プロセスを定義し，これを再利用し，変更要求を正確にプログラムへ反映させる方法について説明する．最後に，本手法の有効性，並びに問題点について議論する．

1.3 Extended MLとML

Extended MLは，MLの拡張言語である．本稿では，Extended MLを仕様記述言語，MLを仕様変更プロセス記述言語ならびに導出されるプログラム言語とする．以下，これらの言語の特徴について説明する．

MLは定理証明システムEdinburg LCF[7]の記述言語として開発された関数型言語である．関数も値として扱うことができるので，高階関数を使った簡潔な表現が可能である[2]．多相型型推論システムであるので，型推論によって信頼性の高いプログラム開発が可能である．さらに大規模なプログラム開発のためのモジュール機構，強力なパターンマッチ機構，例外処理機構をもつ．

関数型言語は関数の定義と関数適用の組合せという簡潔な枠組をもつため，宣言的に仕様を表現することに向いており，実行可能仕様記述言語としても利用できる[12]．型推論機構は，型を仕様と考えると，プログラムの検証に役立つ．しかし，MLのみでは正しいプログラムを開発するための仕様記述言語としては不十分で

ある．

Extended ML は，ML に一階等式論理による公理を追加し，これをプログラムの仕様として，ML のプログラムを段階的に開発するための広範囲言語である．われわれは，厳密な意味定義が行える Extended ML の機構を利用し，型と公理の関係に基づいたモジュール関係を定義することによって，本稿におけるソフトウェア開発のモデルを構築する．要求をシステムの性質として書き下すために，Extended ML の公理の記述を導入することにする．

いわゆる考え方の指針としてだけではなく，仕様変更プロセスを形式的に定義し，利用することが本稿の目的である．そこで，仕様および合成プロセスを ML のデータとして定義し，これらを操作する関数を ML で記述する．1.2 節で述べたように，類似性を損なわないために，これらは同じ言語で記述されなければならない．ML は定理証明システムの定理証明戦略を記述したことでもわかるように，高階関数を多用するこれらの関数記述に適していると考えられる．

以上のような理由から，本稿では，Extended ML を仕様記述言語，ML を仕様変更プロセス記述言語ならびに導出されるプログラム言語として議論する．

2. 要求の設計モデルへの写像

2.1 仕様変更要求

仕様変更と一口にいっても，既存の機能の向上，プログラムの効率の改善，新規の機能の追加といったようにさまざまな要求が考えられる．本稿では，以下に示す設計者*の視点で定義された機能の向上を仕様変更として想定し，議論する．まず，設計者の視点をつぎの設計モデルとして定義する．

[定義 1] 【設計モデル】

システムを複数の主体と環境によってモデル化する．ここで，主体とは，システムにおいて能動的に行動する対象である．環境とは，主体と相互作用を及ぼし合うものとして見た外界であり，複数の対象のネットワークによって構成される．主体の機能とは，「主体がある環境では〜な振舞いをする」ことである．

なお，主体の振舞いおよび環境の行う処理をアクション，アクションの対象をデータと呼ぶ． □

[定義 2] 【機能の向上】

主体 A の機能の向上は，つぎのように変化した環境における A の振る舞いを定めることで達成される．

（1） 既存の環境における A のアクションの対象であるデータの一部が詳細化され，詳細化されたデータに関して，A のアクションに対する新たな制約条件が追加された環境．データの詳細化には，つぎの 2 つの場合がある．

（a） 既存のデータの部分構造を定義する．

（b） 既存のデータの新たな場合わけを定義する．

（2） 既存の環境とは独立なデータが環境に追加され，このデータに関して A のアクションに対する制約が追加された環境． □

上述の定義を例で考えてみる．つぎのような方針で設計された仕様を満たすプログラムが定義されているとする．

仕様：

1 台の自動車が，片側 1 車線道路の交差点を右折し，目的方向へ走行することをシミュレートする．

設計方針：

- 主体を≪自動車≫とし，≪道路≫という環境で，アクション≪走行する≫を行う，というモデルを想定する．要求文はつぎのように書ける．下線が，主体・環境・アクションを表す．

> 1 台の自動車が片側 1 車線道路の交差点を右折し，目的方向へ走行する．

- 交通規則に従って，右折をアクションの制約として定義する．
- 道路にもさまざまな形状があるので，1 つの交差点のみを含む道路をここでの環境とする．

本稿の目的は，仕様変更要求によって変化した環境における自動車の走行の仕様をはじめの自動車の走行の仕様と関連づけて定義し，前者の合成プロセスを利用してプログラムを修正して，各要求に対応する自動車の走行機能の向上を実現することである．仕様変更要求と，その時の環境の変化は，例えばつぎのようになる．ここで，四角形で囲まれた文が，仕様変更要求であり，番号は上記の定義 2 内の番号と対応する．下線がはじめの仕様からの変更部分である．

(1)(a)

> 1 台の自動車が片側 2 車線交差点を右折し，目的方向へ走行する．
> ただし，右折する前に，あらかじめ右側の車線に車線変更を行うものとする．

はじめの環境における走行では，道路上の車線を考慮する必要はなかった．道路のデータの部分構造として車線が追加される．さらに，走行のアクションに対して，車線変更の制約が追加される．

* 1 章では開発者と呼んでいたが，以下設計者とする．

(b) 1台の自動車がロータリーを右折し，目的方向へ走行する．

ロータリーは通常の交差点とは異なる形状をもつ．交差点は，データ「道路」の場合分けの1つであった．ここではデータ「道路」の別の場合わけであるデータ「ロータリー」を定義する．右左折に関わらず，時計回りに走行するという制約が追加される．

(2) 複数の自動車が交差点を右折し安全に目的方向へ走行する．

はじめの環境においては，1台の自動車の走行を考えていた．複数台が走行する環境に拡張するために，他の自動車の状況というこれまでの環境とは独立した情報が追加される．新たな環境においては，他の自動車の状況を見ながら安全に走行するための制約（右折する時に対向車が接近していれば止まる等）が追加される．

2.2 Extended MLによるモジュール設計

まずはじめに，Extended MLで表現された設計仕様を定義する．Extended MLは，モジュールインタフェースにはMLのシグニチャと公理を記述し，モジュール本体にはMLの関数の代わりに公理を記述する言語である．段階的にMLのプログラムへと詳細化するステップの意味が定義された，厳格なソフトウェア開発言語である．公理は，一階等式論理におけるブール式であり，結合子 not, andalso, orelse, ⇒と限定記号 exist, forall を使って構成される．

［定義3］ 【設計仕様】

設計仕様とは，設計モデルにおける主体・環境・データ・アクションをつぎのように *Extended ML* で表現したものである．

主体および環境を構成する複数の対象は，各々 *Extended ML* のモジュールである．主体を表すモジュールを主体モジュール，環境を構成する各モジュールを環境モジュールと呼ぶ．データおよびアクションは *Extended ML* の個々のデータ・関数および公理で定義される．各モジュールはモジュールインタフェースとモジュール本体で構成される．

- モジュールインタフェースは signature で宣言され，つぎの定義を含む．
  - —type および val で宣言されるデータ・アクションの型の定義．
  - —axiom で宣言される主体および環境のアクションの公理*．
  - —axiom で宣言される主体の環境におけるアクションの公理．
- モジュール本体は structure または functor で宣言され，つぎの定義を含む．
  - —datatype で宣言されるデータのデータ構造定義．
  - —fun で宣言される環境のアクションの関数定義．
- モジュールはその他に属性として派生関係（予約語 relation）と差分（予約語 difference）をもつ**．

ここで，signature・structure・functor・type・val・axiom・datatype・fun は，*Extended ML* の予約語である．ただし，structure はパラメータをもたないモジュール，functor はパラメータをもつモジュール本体を定義する． □

つぎに，設計仕様のデータ仕様とアクション仕様を定義し，その設計方法について2.1節で述べた自動車の走行シミュレーションの例を用いて説明する．付録Aは定義された設計仕様の一部である．

2.2.1 データ仕様の設計

［定義4］ 【主体および環境モジュールのデータ仕様】

主体および環境モジュールのデータ仕様は，つぎの構文で定義される *ML* の一般型の内，トップレベルにおいては直和関係を含まない型である．

$$\begin{aligned}&\texttt{datatype}\ \alpha_1..\alpha_m\ t = Con_1\ \texttt{of}\ \{t_{11}, \cdots, t_{1k_1}\}\\ &\qquad\qquad\qquad\qquad\quad |.....\\ &\qquad\qquad\qquad\qquad\quad |Con_n\ \texttt{of}\ \{t_{n1}, \cdots, t_{nk_n}\}\end{aligned}$$

ここで，datatype はデータ型定義を表す *ML* の予約語，α_i は型変数，Con_i は値構成子，t および t_{ij} は型構成子を表す．| は，型の直和関係を表す．直和関係に対し，値構成子で関係づけられた各型構成子 $t_{ij}(j=1..k_i)$ の関係を型の直積関係と呼ぶ．ここで，各 t_{ij} も一般型である． □

例えば，自動車の例において，「道路」というデータをつぎのように考える．**図2**に示すように，一本の道は複数のブロックの連なりであり，道路はこれらの集合である．ブロックが重なっている部分が交差点であり，各ブロックは2つのゾーンに分かれている．そこで，環境モジュール「道路」は**図3**のような，部分と全体の関係によって構造化されたデータ仕様をもち，付録Aの datatype road のように Extended ML で

---

* 他のモジュールのアクションには，Modulename.action のように，モジュール名が拡張子として付けられる．これによって，環境のアクションと環境における主体のアクションが区別できる．

** これらは本稿における予約語である．

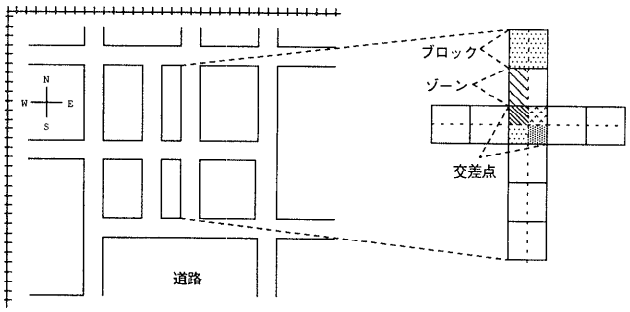

図2 道路のデータ仕様設計
Fig. 2 Design of data structure.

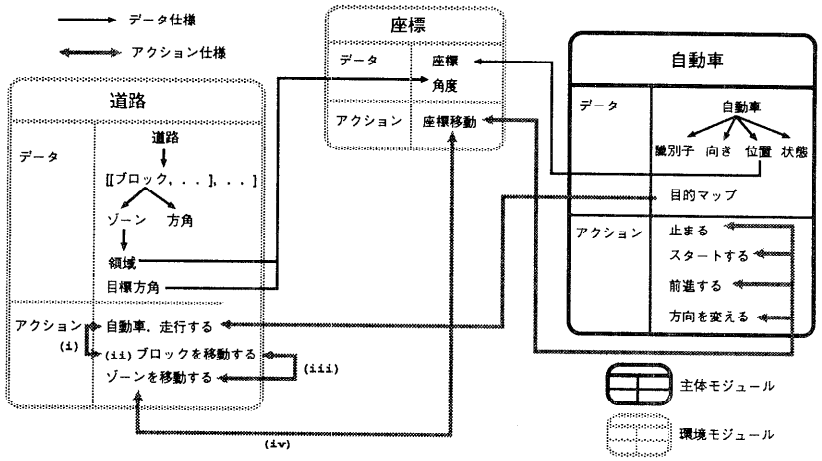

図3 モジュール設計—自動車の例
Fig. 3 Module design—car example.

定義される．

2.2.2 アクション仕様の設計

アクション仕様は，主体のデータ・アクションと環境のデータ・アクションの間に成り立つ関係を一階等式論理の公理として定義したものである．仕様の定義方法にはつぎの2通りがある．

（1） 環境内でのアクション（またはデータ）間の関係を論理式として記述する．

（2） アクションをより抽象的なモデルにおけるアクションとの論理式によって表現する．

付録Aのコメント*(i)～(iv)は，主体≪自動車≫の環境≪片側1車線道路≫における振る舞いを，環境のデータ仕様に基づいて定義する方針を示している．また，図3は，環境「道路」「座標」と主体「自動車」の関係を図示したものであり，矢線はデータ仕様とアクション仕様の構成を示している．

* (*…*)がコメントを表す．

付録Aの定義において，最後の公理（iv）は，ゾーンの移動を，道路のもつ位置と方角の情報を抽象化した「座標」というモジュールをモデルとして定義したものである．その他の公理は，「道路」という環境内で定義された論理式である．

2.3 モジュール関係

仕様変更要求を既存の設計仕様における主体の環境の変化によって実現する．2.1節で定義した環境の変化を，既存の環境モジュールから新しい環境モジュールを定義することによって表す．この新旧2つのモジュール間の関係を**モジュール関係**と呼び，以下これについて説明する．

2.3.1 モジュール関係を考えるとは

Extended ML の functor は，モジュールを生成するモジュールであり，適当なモジュールをパラメータとして与えることによって，新しいモジュールを生成することができる．すなわち，既存の環境モジュールを functor として定義し，変更要求を満たすように新

たなモジュールをパラメータとして与えて，目的のモジュールを生成することもできる．しかし，ここで問題なのは，変更要求を満たすように常に環境モジュールからパラメータを分離できるかということである．例えば，2.1節で定義した3種類の環境の変化を考えてみると，パラメータを変更するのではなく，パラメーターの数を増やしたり，functorの定義自身を書き換える必要があり，決まった型のパラメータを要求される現在のExtended MLのモジュール機構では，これらの変化を記述することはできない．

また，本稿の目的である仕様変更要求をプログラムに反映させるためには，新しいモジュールを作成した手順が必要である．そこで，Extended MLのモジュール機構で定義できるモジュール関係も含めて環境の変化を実現するためのモジュール関係を分類し，定義する．

2.3.2 モジュール関係の分類

モジュール関係は，既存のデータ仕様をどのように変更するか，データ仕様の変更に伴い，どこにどんなアクション仕様を追加するかということで特徴づけられる．まずはじめに，データ仕様の変更について説明する．

データ仕様は型である．モジュールはこの型をもつ値の集合であると考えられる．モジュールを変更して，部分集合を定義する方法には，

1. 型のサブタイプをとる
2. 値を条件式で区別する
3. 値を指定する

がある．2および3は型は不変であるが，はじめのアクション仕様の型に対応する値を制限することによってアクションのプログラムを変更する場合に相当する．紙面の都合上，ここでは言及しない．本稿では，型の構成[3]の考え方に基づき，データ仕様の変更を旧データ仕様Aと新データ仕様B間の関係において，つぎの4つに分類する．

（1） A直積関係B：Cardelliの導入した型集合上のサブタイプ関係[3]であり，つぎの2種類がある．

（I） データ間に直列に新しいデータを挿入する．例えば，つぎのzoneとnewzoneの関係である．この関係を直積直列挿入と呼ぶ．

```
datatype zone=Zone of {region: reg}
datatype newzone
  =Zone of {lanes: lane list}
datatype lane=Lane of {region: reg}
```

（II） データ間に並列に新しいデータを挿入する．例えば，つぎのzoneとnewzoneの関係である．この関係を直積並列関係と呼ぶ．

```
datatype zone=Zone of {region: region}
datatype newzone
  =Zone of {roadsign: sign, reg: region}
```

（2） A直和関係B：例えば，つぎのsectionとnewsectionの関係である．

```
datatype section=Main station station
datatype newsection=Main station station
                  | Local station station
```

（3） A独立B*：例えば，付録Aのdatatype roadとつぎのstate_of_othersの関係である．ここで，carは，環境ROADにおける主体のデータである．

```
datatype state_of_others
  =State of {others: car list}
```

（4） A再帰構造B：例えば，つぎのroadとroadsの関係である．Aの構造を再帰的構造にする．

```
datatype road
  =Road of {streets: block list}
datatype roads
  =Road of {streets: (block list) list}
```

上記のデータ仕様の変更に基づきアクション仕様の変更を整理して，モジュール関係を分類する．

ここでモジュール Mod_A のデータ仕様はつぎの型 t_A とする．作成されたモジュールを Mod_B とし，設計者が定義する Mod_A と Mod_B の差異を**仕様の差分**と呼ぶ．

$$\text{datatype } \alpha_1..\alpha_m\ t_A = Con_A \text{ of } \{t_{A1}, \cdots, t_{Ak_1}\}$$

設計仕様を構成するモジュール間の関係には，Extended MLのモジュール機構によって定義できる関係がある．すなわち，$Mod_A \xrightarrow{relation} Mod_B$ は，Mod_A をfunctorとし，Mod_B をその実パラメータとする関係である．本稿では，モジュール関係で定義された仕様の差分を根拠として，これを利用して合成プロセスを再利用する手順を決めるので，上記以外のデータ仕様の変更に依存する類似関係のみを議論する．以下のモジュール関係を特に**モジュールの類似関係**と呼ぶ．

（i） 継承関係：

● ある1つのデータ型 t_{Ai} と

* Aは環境のデータ仕様であり，主体のデータ仕様は独立に定義されている．ここでは，環境を変化させる要因が主体であり，環境内の主体が1からn(n>=2)に変化して，環境Bとなっている．

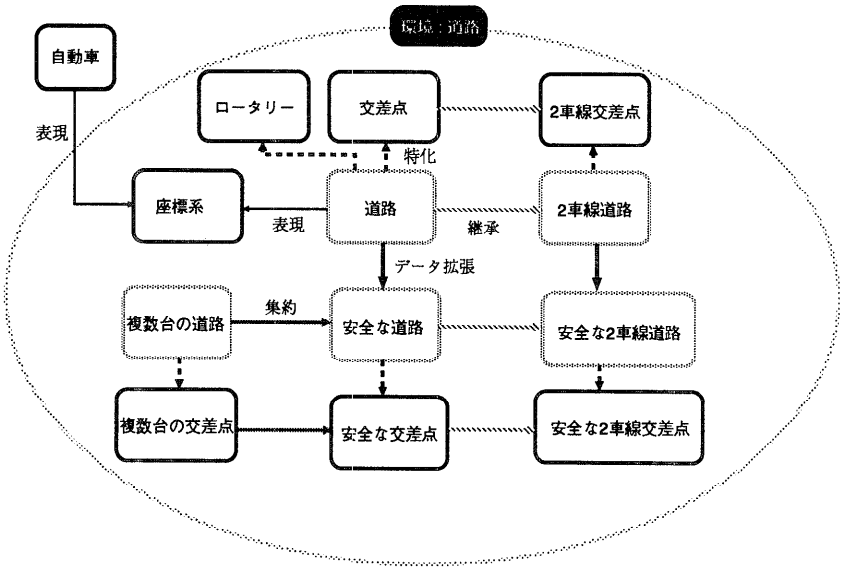

図4 自動車の環境
Fig. 4 Environment for car.

(1)(I) 直積直列挿入
(1)(II) 直積並列挿入
(2) 直和

の関係で，新たなデータ型 t_{Bk} を追加する．あるいは，データ仕様は不変である．

● Mod_A のアクション仕様に，t_{Bk} と t_{Ai} の関係および，t_{Bk} に関する新たな制約を公理として追加する．

【例】2.1の例(1)(a)．

(ii) データ拡張関係：

● t_A と独立なデータ型 s を定義する((3)独立関係)．

● s を Mod_A のアクション仕様の入力として，各アクションのインタフェースを拡張する．s と各 t_A, t_{Ak_i} の間の新たな公理を追加する．

【例】2.1の例(2)．

(iii) 構造拡張関係：

● 初期型を導入し，t_A を再帰構造に拡張する((4)再帰構造関係)．

● Mod_A のアクションは不変であり，新たにアクションの再帰的構造を定義する．

機能の向上は，モジュール関係によって実現される．2.1節で述べた例におけるさまざまな環境の変化によって，**図4**のように環境が変化し，様々な自動車の走行が実現される．ここで，四角形が各モジュールを，矢線がモジュール関係を表している．図4に登場する類似関係以外のモジュール関係について，例で簡単に説明する．

図2のように，ブロックの連なりの集合として一般的に定義された「道路」のデータに対し，1つの交差点のみを含む道路データを考える．これは，型「道路」の値を特定のデータパターンとして表現した値の指定(2.3.2項参照)であり，データの型そのものは変化していない．この関係が特化関係である．図4において，外枠が薄い四角形を特化したモジュールが，外枠の濃い四角形である．

道路は形状とともに領域や方角という属性をもっている．自動車も位置という属性をもっている．これらの位置情報を表すデータとして座標を考える(図3参照)．座標は自動車と道路のアクションを関係づけているが，このアプリケーションに依存しないモジュールとして用意できる．このような抽象モデルを表すモジュールを利用する関係が表現関係である．

2.1節の例(2)の要求を実現するためには，(ii)のデータ拡張関係によって，各自動車が他の自動車の状況を見て安全に走行できる環境を定義しただけでは不十分である．複数の自動車が同期をとって走行する環境を定義する必要がある．データ拡張関係によって定義された環境モジュールにおける自動車のアクションは不変とし，入力となっている「他車の状況」が変化するタイミングを規定する公理を新たに導入する関係が集約関係である．

なお，継承や拡張関係は，値の特定である特化関係とは可換である*.

2.4 モジュール操作

モジュール関係を使って，設計仕様を変更する手段を定義する.

［定義5］ 【モジュール操作】

モジュール関係を使って，仕様変更要求を設計モデルに組み込むための手段をモジュール操作と呼ぶ.

モジュール操作は，つぎに示す *ML* 記述のモジュール操作関数の組合せで定義される.

●モジュール操作関数の入力は，*Extended ML* で記述された仕様を *Extended ML* の構文規則で構造化して表現した *ML* のデータであり，つぎの型をもつ.

```
datatype module
=Module of {definition: definition,
  subjects: subject list,
  relation:(module_name, relclass) list,
  difference: difference}
```

●モジュール操作関数は，この *ML* のデータを操作する関数の集合であり，つぎの種類の関数を含む**.

—環境の操作：環境における環境モジュールの検索や，環境へのモジュールの追加を行う.

—モジュールの操作：モジュールの定義をコピーして新しいモジュールを生成したり，データ仕様やアクション仕様の追加・削除およびモジュールの属性の更新等を行う.

—データの操作：データ仕様やアクション仕様において，データパターンの変換・公理の前提部への論理式の追加等のデータ変更を行う. □

2.5 仕様の差分定義プロセス

仕様変更要求は，Extended ML の設計仕様におけるモジュール関係を設計仕様の加工の根拠とし，モジュール操作を手段とすることで，設計仕様に反映される．つぎに，実際に仕様変更要求を設計仕様に反映させる手順を決定する．この手順が1.1節で述べた**仕様の差分定義プロセス**であり，根拠であるモジュール関係ごとに定義される.

仕様の差分定義プロセスは，変更対象モジュールと設計者によるデータ仕様とアクション仕様の変更定義項目を入力とし，前述のモジュール操作関数のみを使って，変更対象モジュールを修正する手続きである．本プロセスは ML の関数であるモジュール操作関数を組み合わせた ML の関数として定義される．記述は紙面の都合上省略する．付録 B は付録 A のモジュールに対し，継承関係によるモジュールを定義する場合に設計者が与える定義の一部である.

基本的な手順はつぎのとおりであり，以下に示すデータ仕様の差分 DATA とアクション仕様の差分 AXIOM をインタラクティブに入力して，変更が行われる***.

● DATA は，

（a） 追加されるデータの型宣言

（b） 追加データの型定義

（c） 追加方法（trans_type（D_series）等）

で定義される.

● AXIOM は，

（i） 追加されるアクションの型宣言

（ii） 追加公理

（iii） 追加方法（add_data Independent 等）

で定義される.

（1） 変更対象モジュール *Mod* のモジュールインタフェース *Sig* およびモジュール本体 *Body* をモジュール操作関数を使ってコピーする．コピーされたモジュールインタフェースおよびモジュール本体を *Sig'*, *Body'* とし，各々属性〈派生関係〉を更新する.

（2） モジュール操作関数を使い，*Sig'* および *Body'* を DATA に従って変更する.

（3） モジュール操作関数を使い，（2）で変更された *Sig'* に各 DATA の型宣言（a）および各 AXIOM のアクションの型宣言（i）を追加する.

（4） モジュール操作関数を使い，（3）で変更された *Sig'* を AXIOM に従って変更する.

（5） 変更された *Sig'* と *Body'* から，make_module により，目的のモジュールを生成する.

以下，継承関係（データ仕様の直積直列挿入）の仕様の差分定義プロセスにおけるデータ仕様とアクション仕様の修正手順を例を用いて説明する．2.1節で述べた仕様変更要求を満たす設計仕様を，モジュール「道路」から継承関係によって作成する．設計者が定義する設計仕様は，付録 B のようになる．下記の番号は，付録 B のコメントの番号と対応する．また，()内の記号は，付録 B の仕様における関数名である．詳しくは付録 B のコメントを参照のこと.

---

* 可換とはつぎのことである．モジュール A から継承関係で B を定義し，それを特化したモジュールと，A から同じ特化関係で定義した C から同じ継承関係で定義したモジュールは同じである.

** ML で記述を行ったが，紙面の都合上省略する.

*** 以下の trans_type, add_data はモジュール操作関数であり，D_series, Independent はモジュール関係を表す定数である.

(i) 「道路」のデータ構造の部分構造である「ゾーン」の部分構造として「車線」を追加する*. *Body′* に対し，データ仕様の変更をつぎの手順で行う．

(a) *Body′* の datatype 宣言から DATA の追加データの型宣言(a)zone と同名の型定義を検索する．検索された型定義を oldtype，置き換える型定義(b)を newtype と記す．

(b) モジュール操作関数 make_pattern により，oldtype, newtype に対応する変数パターンを生成する．それぞれ，oldpat, newpat と記す．

(c) *Body′* における oldtype をモジュール操作関数 replace_exp により，newtype に置き換える．DATA 内のその他の追加データの型定義をモジュール操作関数 add exp により *Body′* に追加する．ここでは，「車線」の定義 lane が追加される．

(ii) *Sig′* に対し，アクション仕様の変更をつぎの手順で行う**．

(a) *Sig′* の各公理に対し，trans_exp により，oldpat を newpat で置き換える．

(b) oldtype を入力に含むアクション名を *Sig′* の型宣言から検索する．検索されたアクション名を action と記す．ここでは，zone_action が検索される．

(c) action を含む公理を *Sig′* から選択する．ここでは，zone_action に関する公理が選択される．

(d) データを直積直列関係で挿入したので，この公理に置き換わる公理を入力する．「ゾーン」(全体) における主体のアクション (zone_action) を「レーン」(部分) におけるアクション (lane_action) に翻訳する関係を定義する．連続するゾーンを a_1, a_2 とする．各ゾーンは，n 個のレーン b_{ki}(k=1, 2)(i=1..n) で構成され，右折の場合には，ゾーン a_1 から a_2 への移動がある b_{1i}(i=1..n) から，b_{2i+1} への移動になる．

(e) 入力された公理を，モジュール操作関数 replace_data によって選択された zone_action の公理と入れ換える．

(f) 追加アクションのアクション公理をモジュール操作関数 add_data Independent によってモジュールインタフェース *Sig′* に追加する．ここでは，ゾーンの移動と座標の移動の関係式と同様に，車線変更をレーンの移動と座標の移動との関係 (trans_l_real) として定義する．

3. プログラムの合成

3.1 合成プロセスの定義

合成プロセスをつぎのように定義する．

[**定義 6**] 【合成プロセス】

一階等式論理の推論規則の適用によって，設計仕様からプログラムを導出する過程をプログラムの**合成**と呼び，その履歴を**合成プロセス**と呼ぶ．

設計仕様は，*Extended ML* で定義された一階等式論理における公理の集合である．プログラムは，*ML* の関数定義である．プログラムを合成する変換規則は，等式論理における推論規則である． □

合成プロセスは，与えられた式に等式論理[1]における推論規則を適用し，書き換えた式を返すサブプロセスから構成される．このサブプロセスを履歴として記録しながら，ML のプログラムが得られるまで，これを繰り返す．そこで履歴は規則が適用された部分式，規則として適用された式(aexp)，適用された規則の根拠となる規則(rule)，追加された定理等，規則の適用結果の組から構成されるサブプロセスの列である．付録 C にサブプロセスの例を示す．rule は置換律などの推論規則であり，aexp に適用することによって具体的な書換えが定義される．例えば，つぎのようになる***．

replace_rule aexp は，型 equation->Spec.exp->equation の関数である．

書き換え対象の equation とその部分式 Spec.exp を与えると，置換律によって equation が得られる．

3.2 合成プロセスをまねる

本稿では，合成プロセスを再利用することを**合成プロセスをまねる**という．合成プロセスをまねるためには，まねるための根拠・手段・手順を定義しなければならない．

本稿で述べたモジュール関係は類似性を定義する関係であることから，仕様の差分定義プロセスは，各モジュール関係を根拠に，設計仕様を対象としたモジュール操作を手段として，仕様をまねる手順であると言える．そこで，合成プロセスをまねることも，仕様の差分定義プロセスの定義にならって定義する．

---

* これが前記(2)の *Body′* の変更手順である．

** これが前記(2)(3)(4)の *Sig′* の変更手順である．

*** 以下において exp, equation, axiom, signature 等は，データ化された設計仕様の型を表す．

合成プロセスをまねるとは，仕様の差分定義プロセスを根拠として，合成プロセスを対象としたプロセス操作を手段として定義し，それを用いる手順であると考える．プロセス操作は，モジュール操作と同様にプロセス操作関数の組合せとして，3.2.2項で定義する．

3.2.1 まねるの形式的定義

まずはじめに，モジュールの類似関係の仕様の差分定義プロセスにおいて，どの情報が合成プロセスをまねる根拠として具体的に利用できるのかを考える．この情報を**仕様のまね方**と呼ぶ．

ここで，まね方を形式的に定義しよう．

まねるとは，まねる対象を，ある根拠を利用して，ある手段によって加工することである．

そこで，仕様変更プロセスにおけるまね方とは，ある対象をまねる時に，まねた対象に依存した具体的操作であると考えられるので，つぎのように定義する．

［**定義7**］ 【まね方】

$$\left\{\begin{array}{c}\text{設計仕様}\\ \text{合成プロセス}\end{array}\right\}\text{のまね方}$$

$$\equiv apply\left\{\begin{array}{c}\text{モジュール操作関数}\\ \text{プロセス操作関数}\end{array}\right\}\begin{array}{l}\text{根拠から生成さ}\\ \text{れる具体値}\end{array}$$

□

ここで，設計仕様のまね方はモジュール操作関数と仕様の差分の具体値で定義されるので，モジュールの類似関係の種類には依存しない．仕様のまね方は，つぎのとおりである．

●型に対応する変数パターンの変更のモジュール操作関数を

trans_exp: d_class->exp->exp->axiom->axiom とした時に，

trans_exp D_series oldpat newpat: axiom->axiom が1つのまね方である．

これを**データ仕様のまね方**と呼ぶ．ここで，oldpat, newpat はデータ仕様に現れる変数パターンの具体値である．また，d_class はデータ仕様の変更の種類を表す型であり，D_series 等の値を持つ．

●公理の入れ換えに関するモジュール操作を

replace_data: axiom->axiom->signature->signature とした時に，

replace_data oldaxiom newaxiom: signature->signature が1つのまね方である．

これを**アクション仕様のまね方**と呼ぶ．ここで，oldaxiom, newaxiom は入れ換え公理の具体値である．

このように定義すると，合成プロセスのまね方をまねて，目的の設計仕様に適用する合成プロセスをつくることも考えられる．

3.2.2 プロセス操作

［**定義8**］ 【プロセス操作】

合成プロセスをまねる手段をプロセス操作と呼ぶ．プロセス操作は，つぎに示す *ML* 記述のプロセス操作関数の組合せで定義される．

●プロセス操作関数の入力は，3.1節で定義した合成プロセスを，その形式で構造化した *ML* のデータであり，つぎの型をもつ．

```
datatype process=
Proc of {subexp: Spec.exp,
  aexp: Spec.exp, aprule: rule,
  th: theorem, newexp: Spec.exp}
```

●プロセス操作関数は，この *ML* データを操作する関数の集合であり，つぎのような関数を含む*．

—プロセスの操作：設計仕様に対応する合成プロセスデータを蓄積したり，検索を行う．

—サブプロセスの操作：部分式，推論規則，等式などサブプロセスのデータの検索や更新を行う．

—合成操作：書換え規則の生成・適合する公理の検索などサブプロセスを生成するための操作である．

—プロセスの変更操作：仕様のまね方を合成プロセスに適用する操作である．

—再合成の操作：再合成を開始し，その結果を記録する操作である． □

3.3 合成プロセスをまねるプロセス

仕様のまね方は，モジュール操作関数で定義されるので，モジュールの類似関係の種類に依存しない．しかし，プロセス操作を手段として，仕様のまね方を合成プロセスに適用する場合には，モジュール関係ごとに異なる手順が定義される．これを**合成プロセスをまねるプロセス**と呼ぶ．ここで仕様のまね方が合成プロセスをまねる根拠となる．

合成プロセスは，モジュールに定義された公理・関数・ライブラリ関数・ライブラリ関数の定理および補題・合成中に追加された定理によって構成されている．モジュール関係によって設計仕様が変更された部分は，モジュールに定義された公理のみである．そこで，合成プロセスをまねる場合には，モジュールに定義された公理以外の書換え操作は再利用できる．

継承関係（直積直列挿入）における合成プロセスをまねるプロセスを例を示しながら説明する．他の関係におけるまねるプロセスについては説明を省略する．

［**前提**］アクション *action* が，環境モジュール Mod_A

* モジュール操作関数と同様にMLで定義されるが，詳細は省略する．

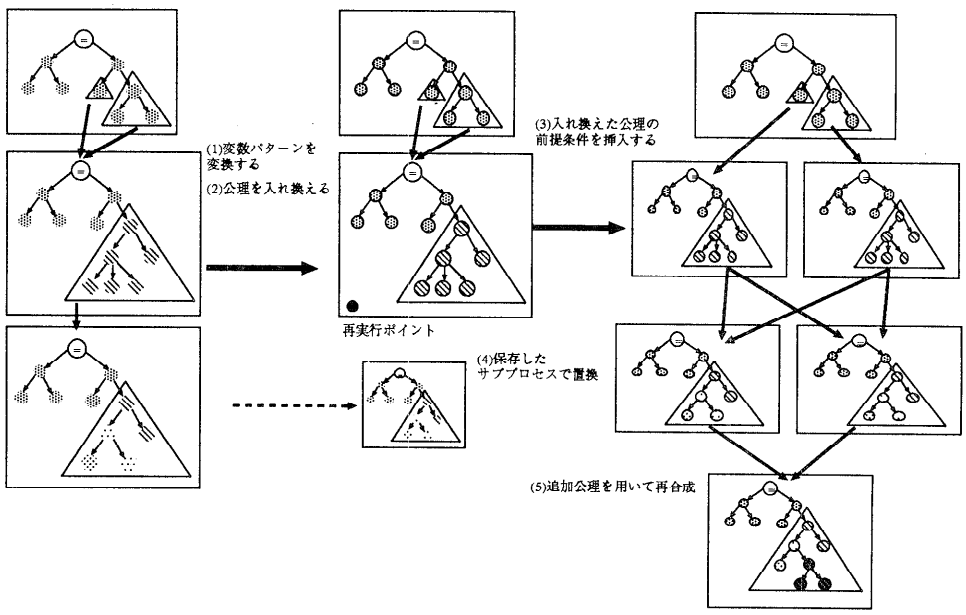

図5 合成プロセスをまねるプロセス
Fig. 5 Process of imitating synthesis process.

において合成され，プログラム $prog_A$ が生成された．この時の合成プロセスを $Proc_A$ と記す．

【例】自動車が道路を走行する．付録Aにおける，*action*＝「走行する（'drive'）」，Mod_A＝「道路（'ROAD'）」，$prog_A$＝'fun drive'.

[**仕様変更**]仕様変更要求を分析し，$Mod_A \overset{relation}{\longrightarrow} Mod_B$ というモジュール関係で，Mod_A から，モジュール Mod_B を作成する．ここで，relation は類似関係である．Mod_B における *action* のプログラム $prog_B$ が仕様変更要求を満たすようにプログラムを合成する．

【例】自動車が2車線道路上を走行する．規則に従って車線変更もできる．付録Bにおける，Mod_B＝「2車線道路（'NLANEROAD'）」，$prog_B$＝'fun drive'.

$Mod_A \overset{relation}{\longrightarrow} Mod_B$ の関係より，まねる対象のプロセスは，$Proc_A$ である．以下の手続きで，$Proc_A$ をまねて，$prog_B$ を合成する．

（I） 仕様の差分定義プロセスのデータが蓄積されている履歴の値を Mod_B 名と relation で検索し，仕様のまね方のデータを得る．データ仕様のまね方を *data_imitate*，アクション仕様のまね方を *axiom_imitate* と記す．

（II） *data_imitate* を $Proc_A$ に適用する（図5(1)．付録D参照.）．*data_imitate* は，対象データ内に変換に適合するデータがある場合にのみ，変換を行う関数である．

【例】ゾーンのデータを表す変数がレーン構造をもつデータパターンに置き換えられる．

（III） *axiom_imitate* を $Proc_A$ に適用する（図5(2)）．*axiom_imitate* は，対象データ内に公理の置き換えに適合するデータがある場合に置換えを行い，置き換えたサブプロセスを返す関数である．置き換えられたサブプロセスを再実行ポイントとして登録する．

【例】ゾーンでの移動に関する公理が，ゾーンの移動をレーンの移動に翻訳する公理に置き換えられる（2.5節の例を参照）．

（IV） 再実行ポイントから再合成する（サブプロセスを生成する）．
置き換える公理 *axiom* がはじめの公理の論理式 *orig_axiom* 以外の論理式 $subaxiom_i$ を⇒の左辺に含む場合には，それらを前提条件として，つぎのように再合成を行う．

- 論理式 $subaxiom_i$ が，*orig_axiom* が書換えの対象とする部分式 *exp* 1 を含む式 *exp* 2 に対する公理であれば，この部分式 *exp* 1 以降のサブプロセスを保存し，*exp* 2 の再合成を行う（図5(3)）．再合成されたサブプロセスの部分式 *exp* 1 を保存した *exp* 1 以降のサブプロセスで置き換える（図5(4)）．
- 追加公理を用いて，再実行ポイントから再合成を行う（図5(5)）．

再合成された範囲を，使用した公理および部分論理式とともに記録する．再実行のプロセスは合成のプロセスと同様に設計者がインタラクティブにサブプロセスを定義する．

【例】ゾーンでの移動に関する公理は，1つの論理式であったが，置き換えられた公理は，前者を含む複数の論理式で構成されている．

既存の合成プロセスをまねた合成プロセスによってつくられたプログラムが，はじめの仕様を満たすことを，合成プロセスの**正当性**と呼ぶ．ここでは，合成プロセスを変更する任意のプロセス操作関数は，合成プロセスのステップを保存することが成り立つことから，合成プロセスの正当性が言える．

4. 考　　察

4.1 有　効　性

仕様変更に対するわれわれのアプローチは，つぎの点で有効であると考える．

まず，関数型言語を用いたことの効果は，つぎのように考えられる．

●関数をカリー化し，適当な引数を与えると，特殊化された関数が定義できる．本稿ではモジュール操作関数やプロセス操作関数を使って，仕様の差分や合成プロセスをまねることを定義した．これらのプロセスの履歴の中から，モジュールの変更や合成プロセスをまねる時の具体的な値を適用した特殊化された関数として，まね方を自然に定義することでき，そのまま新しい問題に適用することができた．モジュール操作やプロセス操作の簡潔な定義には，このような高階関数の利用が自然で有効である．

つぎに，プロセスを利用したことの効果を考えてみる．

● Extended ML における functor は，モジュールを生成するモジュールである．パラメータのモジュールを変更することで，いろいろなモジュールが定義できるが，本稿で示したようなデータ仕様やアクション仕様の変更手続きを表現することはできない．現在のパラメタライズの機構における変更点のプラスマイナスの解釈は，継承やOBJ[6),8)]のビューにみられるような変数の読み替えである．変更点だけをプラスマイナスするのではなく，変更要求の種類によって，どこを，どのように変更するのかという情報が重要である．そこで，仕様変更を扱うには，本稿のモジュール操作を開発環境のコマンド操作として実現するか，解釈系に組み込んだ言語を設計することが必要である．

●本稿の例では，自動車の走行プログラムが自動車の基本的なアクションの組合せで定義されているのに対し，仕様変更要求は道路のデータに関して与えられる．そこで，仕様変更を満たすようにプログラムを直接変更することは，仕様を変更することに比べて困難である．また，仕様変更をあらかじめ予測して十分抽象化された道路のモジュールを定義することも難しいので，仕様変更要求を正確にプログラムに反映させるためには，合成プロセスの利用が必要である．

最後に，まねることの効果を考える．自然言語処理や誘導類推のアプローチにも見られるように，一般的な方法を規定するのではなく，具体的な情報をできるだけまねて活用することの有効性は認められつつある．本稿においても，まねることで元のプログラムの作成意図を変更せずに容易にプログラムを合成できることがわかった．

4.2 適用範囲と問題点

本アプローチにおける第一の問題点は，適用範囲の問題である．

適用範囲の問題はつぎのようにいろいろなレベルで考えられ，適用範囲を拡大するための課題は多い．

●仕様変更要求レベルにおける問題：

すなわち，本手法がどんな仕様変更に適用できるかということである．本稿では，システム内の既知の機能に対する変更要求を満たすように，仕様と合成プロセスを再利用することによって，プログラムを修正する方法を提案した．別システムにある機能を実現したり，新規の機能を実現するためには，既知の情報との差異を特定する方法，すなわち仕様の類似性を判定する方法を確立しなければならない．これは具体例をまねることの1番目の問題点(1.2節参照)である．仕様の差分定義プロセスだけでなく，仕様の生成プロセスを利用して基準を定義したいと考えている．

●設計モデルレベルにおける問題：

本手法は，等式論理＋型付きλ計算を基礎としているので，仕様の表現力という点において，適用できる問題が，この基礎に適した分野に限定されるかもしれない．論理を拡張することも考えられる．

●効率良くまねる問題：

1つの仕様変更要求が1つのモジュール関係のみによって実現されるわけではなく，既知の複数のモジュール関係によって実現されることがある．このようにすでに各々の合成プロセスをまねた経験が蓄積されている場合には，合成の再実行をせずにプログラムを合成したい．3.2.1項の定義から，合成プロセスのまね方はプロセス操作関数とその時の具体値から成る関数である．この関数をまねることも考えられるが，厳密に

定義するためには，モジュール関係の「重なり」を定義し，まね方をまねるための前提条件を明らかにする必要がある．詳細は他稿で行うことにする．

第2の問題点は，検証の問題である．アクションの公理の定義は，ユーザに任されている．型推論によって，型の整合性は保証できるが，矛盾なく定義されているかについては，本稿では考察していない．

また，仕様変更要求を実現する方法や，プログラムを合成する方法は，一通りではない．本稿では，1つの成功する方法を特定して議論を行った．本手法は，変更方法を特定する手段ではない．選択した方法が他の方法より有効であるかについても考察すべきである．

5. おわりに

類推によるプログラミングのように，形式定義上の記号操作によってのみ，類似性を見つける方法では，どのような仕様変更に使えるのかがわからない．本稿では，5種類のモジュールの類似関係を使って合成プロセスをまねるプロセスを定義し，仕様の差分に基づいた合成プロセスの再利用方法を示した．仕様変更のプロセスを利用することによって，仕様変更要求を満たすようにプログラムを容易に正しく修正することができた．

本稿で関数として定義したモジュール操作・プロセス操作・合成・まねるを計算機上で実行することができる仕様変更プロセスの支援環境を構築することと，本手法の適用範囲の拡大を今後の課題とする．

謝辞 本研究は，産業科学技術研究開発制度「新ソフトウェア構造化モデルの研究開発」の一環として情報処理振興事業協会（IPA）が新エネルギー・産業技術総合開発機構から委託をうけて実施したものである．

参考文献

1) Bachmair, L., Dershowitz, N. and Hsiang, J.: Orderings for Equational Proofs, *Proc. IEEE Symposium on Logic in Computer Science*, Cambridge, Massachusetts, pp. 346-357 (1986).
2) Bird, R. and Wadler, P.: *Introduction to Functional Programming*, Prentice-Hall (1988).
3) Cardelli, L. and Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Surveys*, Vol. 17, No. 4, pp. 471-522 (1985).
4) Carbonell, J. G.: Derivational Analogy, A Theory of Reconstructive Problem Solving and Expertise Acquition, ed. Michalski, R. S. et al., *Machine Learning II : An Artificial Intelligence Approach*, Morgan Kaufmann (1986).
5) Dershowitz, N.: Program by Analogy, cd. R.S. Michalski et al., *Machine Learning II : An Artificial Intelligence Approach*, Morgan Kaufmann (1986).
6) Futatsugi, K., Goguen, J., Meseguer, J. and Okada, K.: Parameterized Programming in OBJ2, *Proc. 9th Int. Conf. on S. E.*, pp. 51-60 (1987).
7) Gordon, M. J., Milner, A. J. and Wadworth, C. P.: Edinburgh LCF, *LNCS*, Vol. 78, Springer-Verlag (1979)
8) Gouguen, J. A.: Principles of Parameterized Programming, ed. Biggerstaff, T. J. et al., *Software Reusability, Vol. I, Concepts and Models*, Addison Wesley (1989).
9) Goldberg, A.: Reusing Software Developments, *Proc. 4th ACM SIGSOFT Symposium on Software Development Environment*, Vol. 15, pp. 107-119 (1990).
10) Kolodner, J. and Riesbeck, C.: Case-Based Reasoning, *IJCAI-89 Tutorial-MA2* (1989).
11) Lue, J. and Xu, J.: Analogical Program Derivation Based on Type Theory, *Theoretical Computer Science 113*, pp. 259-272 (1993).
12) 松浦，大林：ソフトウェア開発環境 dmCASE，情報処理学会論文誌，Vol. 31, No. 7, pp. 1091-1103 (1990).
13) Milner, R.: Proposal for Standard ML, *Conference Record of 1984 ACM Symposium on LISP and Functional Programming*, ACM, pp. 184-197 (1984).
14) Mostow, J.: Design by Derivational Analogy: Issues in the Automated Replay of Design Plans, *Machine Learning*, pp. 119-184, MIT Press (1990).
15) Paulson, L. C.: *ML for the Working Programmer*, Cambridge University Press (1991).
16) Partsch, H. and Steinbruggen, R.: Program Transformation Systems, *ACM Computing Survey*, Vol. 15, No. 3, pp. 199-236 (1983).
17) Sannella, D. T. and Tarlecki, A.: Toward Formal Development of ML Program: Foundations and Methodology, ed. Diaz, J. and Orejas, F., *Proc. TAPSOFT'89*, Vol. 2, LNCS, Vol. 352, pp. 375-389, Springer-Verlag (1989).

付　　録

A　設計仕様の記述例

```
(* 以下において，@,::は各々append,consの演算子を表す記号である．'a は型変数，Map・Road など先頭が大文字の単語は値構成子を表す．*)
signature ROAD=
sig
  relation [(COORDINATES,Represent)]
   type roadmap
   type road
   type block
   type singleblock
   type zone
   val drive: Car.car -> roadmap -> Car.car list
   val move: Car.car -> Car.compass -> block -> (Coord.zone coord) list
   val position: Car.car -> Coord.real coord
   val zone_action: Car.car -> Car.compass -> block -> Coord.zone coord -> Coord.zone coord -> Coord.real coord
   val trans_real: Car.car -> Car.compass -> block -> (Coord.zone coord) list -> (Coord.real coord) list ...
(* (i) 自動車の走行(drive)を交通規則が定義された道路上の移動アクション(move)によって定義する．*)
(* 最初は初期位置ブロック'initblock'を含むストリートを直進している．*)
   axiom as=map Car.position (Car.drive (init c initblock)(Map {street=ss,init=initblock,objective=finalblock}))
     andalso    bs=map trans_real (concat(map (move c Front)(take_cross (get_street initblock (streets ss)))))
     @[move c (which_direction m)(last(take_cross (get_street initblock (streets ss))))]
     andalso  initpart as bs
(* 最後は目的位置ブロック'finalblock'を含むストリートを直進している．*)
   axiom as=map Car.position (Car.drive (init c initblock)(m as Map {street=ss,init=initblock,objective=finalblock}))
       andalso  bs=map trans_real ((concat (map (move c (which_direction m))
                   (hd(drop_cross (get_street finalblock (streets ss))))))
             :(concat (map (move c Front)(tl(drop_cross(get_street finalblock (streets ss)))))))
       andalso lastpart as bs
(* (ii) 走行車線は一定であるという交通規則を,「ブロックの移動において走行ゾーンは一定である」と定義する．*)
   axiom dir c=a1
 ⇒ move c Right (Cross {first=Single {leftzone=l1,rightzone=r1,direction=a1},
         second=Single {leftzone=l2,rightzone=r2,direction=a2}})
               =[Tuple l1 r2,Tuple l1 l2,Tuple r1 l2]
(* (iii) 自動車のゾーン内で可能なアクションとブロック内の移動アクションの間の制約を定義する．*)
   axiom forall x.forall y. (member (x,y) (zip (move c d cr) (tl(move c d cr))))
                     andalso (zone_action c d cr x =y)
(* (iv) 自動車のゾーン内で可能なアクション(zone_action)を座標の移動(change_pos)で定義する．*)
   axiom zone_action c Right (Cross {first=c1 as Single {leftzone=l1,rightzone=r1,direction=a1},
    second=c2 as Single {leftzone=l2,rightzone=r2,direction=a2}})
           ::(Single {leftzone=l3,rightzone=r3,direction=a3})::nil)(Tuple l1 r2)
   =Tuple l1 l2
⇒ trans_real(Tuple l1 l2)= Coord.change_pos (step c2) a1 (trans_real (Tuple l1 r2))....
end
functor RoadFun (structure coordinates:COORDINATES)=
struct
  local
   open Coordinate
   subject [Car]
   relation [(Coordinate,Represent)]
  in
   datatype roadmap=Map of {street:road,init:block,objective:block}
   datatype road=Road of {streets:(block list) list}
   datatype block= Single of {single:singleblock} | Cross of {first:singleblock, second:singleblock}
   datatype singleblock= Block of {leftzone:zone,rightzone:zone,direction:angle}
   datatype zone=Zone of {region:'a}
end
(* 合成された片側1車線交差点における自動車の走行プログラム *)
(* 前進する(forward)，方向を変える(rotate)等が自動車の基本的なアクションである．*)
fun drive (init c initblock)(Map {street=Road of {streets=(initblock::ss)@[Cross{first=c1,second=c2}]@rs,
                ts@[Cross{first=c2,second=c1}]@us@[finalblock]},init=initblock,objective=finalblock}))=
  let val n1=iterate #ss (Car.forward) (Car.start c)
      val n2=conti [Car.forward, Car.rotate (direction initblock finalblock),Car.forward] (last n1)
      val n3=iterate #us (Car.forward (last n2))
  in n1@n2@n3@[Car.stop (last n3)]
  end
```

B 継承関係における仕様の変更

```
signature NLANEROAD=
sig
  relation [(ROAD,Inherit),(COORDINATES,Represent)]
  difference
(* (ii)(c) 自動車のゾーンにおけるアクションとレーンにおけるアクションの関係を定義する. *)
(* 右折するならば, 自動車はあらかじめ曲がる方向の車両通行帯にいなければならない. *)
axiom  xs=(map (move c Front) ss)@[move c Right (Cross {first=t, second=u})]
  andalso
   forall s,t.member (s,t) (zip ss (tl ss)) andalso
   zone_action c Front ((s as Single {leftzone=Zone {lanes=l11}),rightzone=Zone {lanes=r11}),direction=a1})
 ::(t as Single {leftzone=Zone {lanes=l21}),rightzone=Zone {lanes=r211}),direction=a2})::nil)
      (Point (Zone {lanes=l11}))
       =Point (Zone {lanes=l21})
  andalso l1=in (Car.position c)(map reg l11)
  andalso lanenumber l1 l11 < length l11
⇒ lane_action c Front ((Single {leftzone=Zone {lanes=l11}),rightzone=Zone {lanes=r11}),direction=a1})
 ::(Single {leftzone=Zone {lanes=l21}),rightzone=Zone {lanes=r211}),direction=a2})::nil)(Point l1)
       =Point (get_lane ((lanenumber l1 l11)+1) l21) ...
(* (iii)(e) 車線変更の規則を座標の移動で定義する. *)
  axiom lane_action c d (s::t::nil) (Point l1)=(Ponit l2)
 andalso lanenumber l1 (lanes(leftzone s))=(lanenumber l2 (lanes(leftzone t)))+1
 ⇒  trans_l_real (Point l2)
  =Coord.change_pos (-(lanewidth s))(0.5*pi-(direction s))(Coord.change_pos 1 (direction s)(trans_real (Point l1)))
end
functor M_Lane_RoadFun(structure coordinates:COORDINATES)=
struct
  local
   open Coordinate
   subject [Car]
   relation [(Road,Inherit),(Coordinate,Represent)]
  in
 difference
(* (i) 直積直列関係によるデータ型の変更 *)
 datatype zone=Zone of {lanes:lane list}
 datatype lane=Lane of {region:'a}
end
(* 片側2車線交差点における自動車の走行プログラム *)
fun drive (init c initblock)(Map {street=Road of {streets=(initblock::ss)@[Cross{first=c1,second=c2}]@rs,
                ts@[Cross{first=c2,second=c1}]@us@[finalblock]},init=initblock,objective=finalblock}))=
  let val n1=iterate 4 (Car.forward)(Car.start c)
      val n1'=iterate (#ss-5) (Car.forward) (Car.diagnal (last n1))
      val n2=conti [Car.forward,Car.forward,Car.rotate(direction initblock finalblock),Car.forward,Car.forward](lastn1')
      val n3=iterate #us (Car.forward (last n2))
  in n1@n1'@n2@n3@[Car.stop (last n3)]
  end
```

C 合成プロセスのサブプロセス

```
部分式              zip [Point l1, Point l2][Point l2]
適用された式        zip [Point l1, Point l2][Point l2]
                        =(Point l1, Point l2)::(zip [Point l2] nil)
規則                replace_rule
結果  zip(move c Front ((Single {leftzone=l1,rightzone=r1,direction=a1})
         :: move c Front (Single {leftzone=l2,rightzone=r2,direction=a2})::nil))
     (tl(move c Front ((Single {leftzone=l1,rightzone=r1,direction=a1})
         ::move c Front(Single {leftzone=l2,rightzone=r2,direction=a2})::nil)))
     = (Point l1, Point l2)::(zip [Point l2] nil)
追加規則  なし
```

D 合成プロセスをまねる

```
(* データ仕様のまね方. ここで, #exp# は, 仕様の ML におけるデータ表現である. *)
trans_exp D_series #(Single {leftzone=l1,rightzone=r1,direction=a1})#
          #(Single {leftzone=Zone {lanes=l11},rightzone=Zone {lanes=r11},direction=a1})# ...
(* 仕様のまね方を取り出して, まねる対象の合成プロセスに適用する *)
apply_pat_diff:(axiom -> axiom) -> process list -> process list
new=apply_pat_diff (trans_exp #(Single {leftzone=l1,rightzone=r1,direction=a1})#
  #(Single {leftzone=Zone {lanes=l11},rightzone=Zone {lanes=r11},direction=a1})#) PROC
```

(平成6年 8 月12日受付)
(平成6年12月 5 日採録)

松浦佐江子（正会員）

1955年生．1979年津田塾大学学芸学部数学科卒業．1982年同大学院理学研究科数学専攻修士課程修了．1985年同大学院理学研究科数学専攻博士課程単位取得退学．同年4月(株)管理工学研究所入社，研究員．以来，ソフトウェア開発環境に関する研究開発に従事．ソフトウェア開発環境および設計方法論におけるフォーマルなアプローチ，関数型言語に興味を持つ．1993年，情報処理学会研究賞受賞．日本ソフトウェア科学会，人工知能学会各会員．現在，情報処理振興事業協会・新ソフトウェア構造化モデル研究本部に出向中．

本位田真一（正会員）

1953年生．1976年早稲田大学理工学部電気工学科卒業．1978年同大学院理工学研究科電気工学専攻修士課程修了．工学博士．同年(株)東芝入社．現在，同社研究開発センターシステム・ソフトウェア生産技術研究所に所属．1989年より早稲田大学非常勤講師を兼任．1991年東京工業大学大学院非常勤講師．主として，ソフトウェア工学，人工知能の研究に従事．ソフトウェアの基礎理論に興味を持つ．1986年情報処理学会論文賞受賞．著訳書「ソフトウェア開発のためのプロトタイピング・ツール」(共著)，「KE養成講座(2)エキスパートシステム基礎技術」(共著)，「オブジェクト指向システム分析」(共訳)など．日本ソフトウェア科学会理事．日本ソフトウェア科学会，人工知能学会，IEEE，AAAI各会員．