構造化オブジェクト指向設計図からの実行可能C++プログラム生成

原田 実[†] 水野 高宏[††] 濱田 奨[††,☆]

本論文では，オブジェクト指向設計図から実行可能なC++プログラムを生成する方式について述べる．実際には，この方式に基づいてプログラムを生成する機能を，我々がすでに開発したsomeと呼ぶ構造化オブジェクト設計環境に追加した．プログラム生成の基となる設計図としては，ヘッダファイルに記述される構造定義を生成するために派生図とオブジェクト図を，メソッド記述ファイルに記述される動作定義を生成するためにイベントトレース図と決定表を用いた．イベントトレース図には，メソッド内の動作系列の制御論理をビジュアルに表す図フィールドと，起動条件や計算内容を文章やC++コードで表現するテキストフィールドを持たせ，さらにコレクションクラスへのアクセスの簡略表現機能を盛り込んだ．また，決定表では，ファイル処理やコレクション処理などのデータ構造に依存した制御を簡潔に表現できる高水準言語LOLAを用意し，特にビジネスプログラムなどの条件分岐の多いメソッド仕様を簡略にかつ視覚的に表すことができるようにした．これらの高次表現からC++プログラムへの展開は，プログラムの骨格を表すフレームクリシェにプログラムの制御や計算の部分を表す部品クリシェを設計図に合わせて編集後埋め込むことで行った．複数の問題への適用の結果，設計図は見やすく簡潔なものであった，また生成されたプログラムはコンパイル後正しく動作した．

Executable C++ Program Generation from the Structured Object-oriented Design Diagrams

Minoru Harada,[†] Takahiro Mizuno[††] and Susumu Hamada[††,☆]

In this paper, the method to generate an executable C++ program from object-oriented design diagrams is described. Actually, the function to generate the program based on this method is added to the structured object modeling environment SOME, which we had already developed. The derivation diagrams and the object diagrams are used to generate the structural definition described in the header file and the event-trace diagrams and the decision tables are used to generate the procedure definition described in the method description file. Especially, in the decision table, the high-level specification language LOLA is proposed to express concisely the control logic according to data structures of the file processing and the collection processing. Especially it enables to express concisely and visually the specification of the method having many conditional branches as in the business program. In the process to generate the C++ program from LOLA expressions, Parts Cliches which express the controls and calculations are customized according to design diagrams and embedded in the Frame Cliche which expresses the skeleton of the whole program. As a result of designing several problems, the design was simple and concise, and the generated programs operated correctly.

1. はじめに

1980年代前半に発表されたオブジェクト指向プログラミング[5),18)]は，“ソフトウェア危機”を救う技術として，ここ数年特に注目され，今や主流になりつつある[12)]．特に，最近では上流であるオブジェクト指向分析/設計の重要性が認識され[15)]，その方法論としてシュレアーメラー法[17)]，コード/ヨードン法[3)]，Booch法[2)]，OMT法[16)]，ヤコブソン法[13)]，FUSION法[4)]，UML法[19)]などが提案され，その優位性が議論され進化・統合に向かっている．

これらの方法論に基づく設計図を作成するためのCASEツールが開発され，実際に利用されている．Ra-

† 青山学院大学理工学部経営工学科
Faculty of Science and Engineering, Aoyama Gakuin University
†† 青山学院大学大学院理工学研究科経営工学専攻
Department of Industrial Engineering, Graduate School of Science and Engineering, Aoyama Gakuin University
☆ 現在，富士ゼロックス株式会社
Presently with Fuji Xerox Co., Ltd.

tional 社の Rose[20]はその代表例である．しかし，これらの市販ツールの問題点は，設計図からプログラムを自動生成できないことである．実際はクラスの属性や関係を記述した構造定義ファイル (*.h) を生成できるが，メソッドが行う処理を記述した動作定義ファイル (*.cpp) を生成できないので，生成されたプログラムを実行することはできない．これでは，作成した設計図からプログラムを作成するのは手作業になり一貫した自動化ができない．80 年代の終わり頃には，手続き型の開発方法において，原田[7]の SPACE，IBM の AD/Cycle，日立の SEWB，富士通の YPS/APG，日本電気の SOFPIA，などのように[8]，高水準で視覚化された固有の設計図から C や COBOL プログラムを自動生成できる CASE ツールが多く存在し広く利用されてきたことを考えると，オブジェクト指向に従った開発方法においても同様の自動化技術を用いることで生産性の向上が期待できる．

一方，研究面でも，オブジェクト指向に基づく設計図からのプログラム生成の自動化の研究例は数少ない．Ali ら[1]は，状態遷移図からの JAVA プログラムの生成を実現しているが，このシステムでは，生成されるコードはオブジェクトの状態を正確に遷移するための制御ロジックのみで，メソッド内でオブジェクトが行う具体的な計算処理を生成できない．また，Shlaer と Mellor[21]は，リアルタイムシステムに対する彼ら独自のオブジェクト指向設計図からメソッドの処理記述も含むコードの自動生成を実現している．しかし，ここでのコード生成は，Code archetype に Switch 文などの制御文と属性名を用いて記述したメソッドにおける処理の制御ロジックの骨子に，Instance database に記述した具体的な変数値や算術式を，生成時に埋め込むことで行っている．したがって，制御ロジックの作成は Code archetype の作成者の技量に依存しており，また仕様の抽象度も低いものである．このように，オブジェクト指向プログラム生成の自動化が難しいのは，オブジェクト指向に適したメソッドの機能を表す高水準で実用的な仕様記述法が確立されていないことが最大の原因である．

本研究の目的は，視覚的で高水準なオブジェクト指向設計図から実行可能な C++プログラムを自動生成することである．この目的のために，すでに我々は，構造化された設計表現法を持つ技法 SOMM (Structured Object Modeling Method) を提案し，その有効性を実証するためにそれに基づく開発環境 SOME (Structured Object Modeling Environment) を構築している[9)~11)]．SOMM の最大の特徴

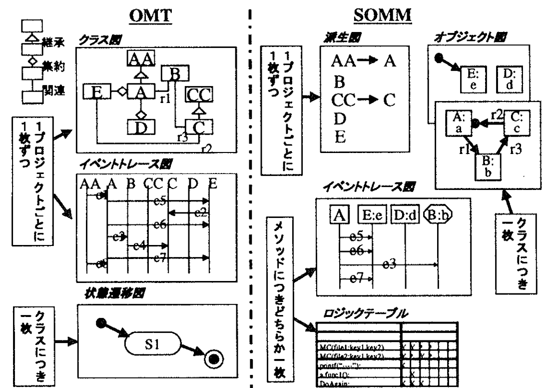


図1 OMT と SOMM の表記法の比較
Fig. 1 Comparison of OMT and SOMM.

は，クラス間の 3 つの関係である関連，継承，集約を，それぞれオブジェクト図，派生図，オブジェクト図間の階層構造に分けて表現することであり，OMT と比較すると図 1 のようになる．このように SOMM には，対象の分析/設計視点として，4 つのモデルが存在する．派生図はクラスの定義とその継承関係を，オブジェクト図はクラスの構成（集約）要素の定義と構成要素間の関連を表現する．また，イベントトレース図には，クラスに送られたイベントに対する反応としての，他クラスへのイベントの送信や計算を記述する．このようにこれまでの SOME では，静的モデルの表現に重点が置かれており，動的モデルを表現するために存在する唯一のイベントトレース図においても一連の単純なイベント送信を表現できるのみで，メソッドが行う具体的な計算やその制御ロジックを表現するにはほど遠いものであった．

そこで今回のプログラム生成の研究のために，SOME の動的モデル表現を大幅に拡充することにした．さらにこの新しいビジュアルな動的モデル表現から実行可能な C ++プログラムを生成できるようにした．具体的には，SOME のイベントトレース図に，新たに反応動作系列中の制御構造を表現する方法とコレクションクラスに対する抽象化表現を加えた．また，条件分岐などが機能仕様の中心的な役割を果たすビジネスプログラム仕様などを記述するために，決定表形式のロジックテーブルを SOME の動的モデルに追加した．さらにロジックテーブルを使ってメソッドの機能を抽象的に記述するための高水準言語 LOLA を開発した．このロジックテーブルと LOLA によるビジネスプログラム仕様の高水準の視覚表現は，第 1 筆者が過去に SPACE というシステムで手続き的な COBOL プログラムの自動生成を行うときに用いたものと同様

マスタファイルには、部No、社員No、基本給、残業単価、前月までの支給合計が記録されている。残業ファイルには、部No、社員No、残業時間が日毎に記録されている。今、以下の条件で今月分の給与を計算し、支給合計を求め、新マスタファイルに書き出す。また、給与リストには、社員名ファイルから得た氏名を含む個人の明細行のみでなく、部毎の合計も出力する。なお、各ファイルは、部No、社員Noに対して昇順にソートされている。
・給与は基本給に残業手当を加えたものである。
・残業手当は残業代を社員ごとに集計したものである。
・残業代は単価×時間×掛率である。
・掛率は、残業区分が1(通常),2(深夜),3(早朝)に応じて、1,1.5,1.2である。

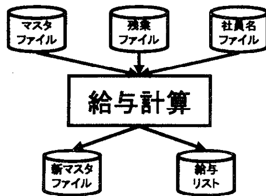


図2 事例としての給与計算問題
Fig. 2 A sample salary problem.

である[7]．したがって，本研究のLOLAに関する部分で本質的なところは，SPACEで用いたような決定表を計算モデルに持つ高水準言語による仕様記述が，オブジェクト指向によるモデル化にも適用できかつ有用なことを実証することである．なお，SOMEにおける設計図の表現法やLOLAを用いた制御の簡易記述法の具体的な理解を助けるために，図**2**に示すような事務処理問題を一貫して例として用いる．

以下では，2章，3章，4章，5章で4つの各設計図の記述法について，また，6章でC++プログラム生成の方式について説明する．結論では，SOMEを評価しその有効性を議論する．

2. 派　生　図

派生図は図**3**に示すような図で，クラスの登録簿である．ここには，プロジェクトに必要なすべてのクラス名とクラス種別を宣言する．クラスは頂点で表記し，C++プログラム生成を目標としているのでクラスとテンプレートを分けて扱うために頂点の形には2種類設けた．1つは通常のクラスを表す頂点で，もう1つはクラス名を○で囲んだクラステンプレートを表すテンプレート頂点である．また，プロジェクト自身も1つのクラスと見なし，2重枠で囲んだ頂点で表し，『新規プロジェクト』をファイルメニューで選ぶと，この頂点だけが存在する派生図が生成される．この頂点の名前を変えることで，プロジェクト名を変更することができる（例：図3では「ProjectX」に変更されている）．

派生図において，あるクラスあるいはテンプレートクラスBを別のクラスあるいはテンプレートクラス

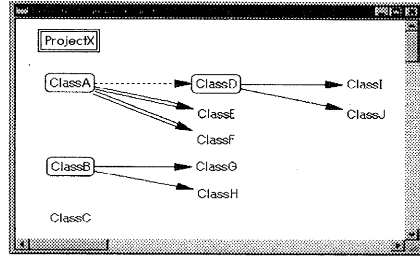


図3 派生図の例
Fig. 3 A sample of derivation diagram.

表1 派生弧の種類
Table 1 Derivation arc types.

派生の種類	表記法	C++での意味
継承	A ——▶ B	class B:A{...};
テンプレート化継承	A ——▶ (B)	template<class T> class B:A {....};
テンプレート継承	(A) ——▶ (B)	template<class T> class B:A<T> {....};
カスタム化	(A) ══▶ B	T→CCの時 #define B A<CC>
カスタム継承	(A) ══▶ B	T→CCの時 class B:A<CC>{...};
展開	A, (A) ⟶ B, (B)	コピー後に編集

Aから，どのような継承あるいはカスタム化（テンプレートを具体化してクラスを作ること）で作成するかを，表**1**に示すような6種類の弧で表す．この表に，それぞれの具体的な作成方法をC++でのクラスBの宣言方法で示した．実際6種類の弧が必要になるのは，頂点A，Bにそれぞれクラスかテンプレートが割り当てられるので2×2の4通り，これに同時にカスタム化と継承を行うカスタム継承と単純な複写である展開を加えて6通りとなる．なお，展開は，Bのオブジェクト図やイベントトレース図をAのそれらを複写して作成することを表しており，複写後の意味的関係は存在しない．

図3の例では，ClassEはテンプレートクラスClassAをカスタム化していること，ClassGがテンプレートクラスClassBを継承していることなどが表されている．

なお，給与計算問題の派生図は図**4**のようになる．ここでは，給与計算問題は8つのクラスから構成され，それらの間には継承関係がないことが分かる．

3. オブジェクト図

SOMEでは，オブジェクト図は，派生図に登録されたクラスを利用者定義のデータ型と考え，その構造を表現するものと考える．したがって，オブジェクト図には，クラスの構成要素であるオブジェクトとこれらオブジェクト間の関連を表現する．

あるクラスC（例：SalaryCalculation）のオブジェ

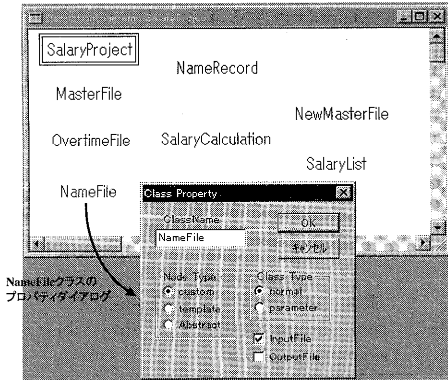


図4 給与計算問題の派生図
Fig. 4 The deribation diagram of the salary problem.

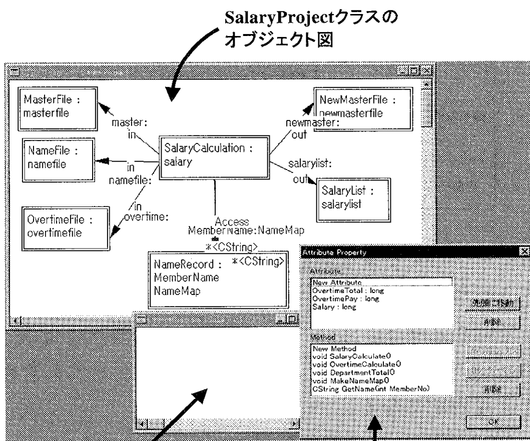


図5 給与問題クラスのオブジェクト図
Fig. 5 The object diagram of the salary problem class.

クト図では，クラスCの諸特性Aを表すint型やchar型などの基本的な型を持つ属性（例：long OvertimeTotal）やメソッドは**図5**に示すようにダイアログから入力し，オブジェクト図には表示しない．

一方，基本型以外の型，すなわち派生図に登録されたクラスSあるいはそのようなクラスへのポインタを型に持つ属性s（例：MasterFile型のmasterfile）があるということを，クラスCとクラスSの間に集約関係があり，CからみたSの役割名（オブジェクト名）がsであると考える．これを，図5に示すように，クラスCに対するオブジェクト図内にオブジェクトs（これをクラスCの子オブジェクト呼ぶ）を表す箱形の頂点を描き，その中にS:sなるラベルを付けて表す．これはグラフ間の関係でいうと，Cのオブジェクト図にSのオブジェクト図を，階層的に結合することを意味している．SOMEではこれを再帰グラフ理論[5]のZoomIn操作で実装している．なお，頂点にはそのラベルとともに，そのオブジェクトインスタンスの多重度（0以上の整数か *）も記述する．さらに，この多重度が * の場合，コード生成時に入れ物クラスの名前を表すコレクション名と，入れ物クラスとしてキーで検索できるMapを使うならキーのデータ型もダイアログから入力する．

クラスC内のオブジェクト図における子オブジェクトs1，s2間の関連は，それらの頂点間の弧で表す．関連には，関連名と役割名（Role名）のほかに，集約と同様，インスタンス多重度（0以上の整数か *）をダイアログから入力する．さらに，多重度が * の場合は，コレクション名も入力する．

なお，zoomInで結合された異なるオブジェクト図内のオブジェクト間の関連は，直接弧を引いて表すことができない．このために，オブジェクト図の内と外をつなぐためにインタフェースを用意し，これにどのような型を持つクラスと結合するかを表すためにクラス名をつける[9]．

図5の例では，SalaryProjectクラスがSalaryCalculationクラスやNameFileクラスを集約していること，SalaryCalculationクラスがMasterFileクラスへのポインタmasterやNameRecordクラスを集めたCStringをキーとするMap型の入れ物クラスへのポインタNameMapなどを持っていること，などが表されている．

4. イベントトレース図

クラスCのオブジェクト図に登録された各メソッド（=内・外イベントに対する反応）に対しては，その機能をイベントトレース図あるいはロジックテーブルのどちらかで表す．

イベントトレース図はクラスCがイベントEを受けたときの反応動作系列を視覚的に表す．この動作の中心となっているのが，「イベント送信」とその起動条件である「制御構造」である．イベントトレース図では**図6**に示すように，「クラスCがイベントEを受け取ったとき，何らかの関係がある他のオブジェクトに対して，どのようなイベントFをどのような条件あるいはタイミングTの下に送信するか」を表現する．

具体的には，図を左右に分割し，左にイベント送信や制御構造を表示する図フィールドを，右にその処理や条件を具体的に表すC++命令や文章（C++のコメントとして）を記述するテキストフィールドを設定する．

図6　GetName メソッドのイベントトレース図
Fig. 6　The event-trace diagram representing the GetName method.

表2　イベントトレース図でのオブジェクトの表記法
Table 2　Notation for objects in event-trace diagram.

表記名	表記	表記の表す意味
セルフオブジェクト	クラス名	イベントトレース図の対象となっているメソッドを所有するクラスのインスタンスを表すオブジェクトである．イベントトレース図のもっとも左側に記述する
子オブジェクト	クラス名:オブジェクト名(コレクション名)	セルフオブジェクトの集約要素を表すオブジェクトで，基本クラス以外の型を持つオブジェクトである．集約要素の多重度が＊のときは，コレクション名が表示される
引数オブジェクト	クラス名 オブジェクト名	イベントトレース図が表しているメソッドの引数となっているオブジェクトである．
局所オブジェクト	クラス名 オブジェクト名	メソッドの内部でローカルに定義され，用いられるオブジェクトである．これは，メソッドの内部で作成され，メソッドの終了とともに破棄されるオブジェクトのことである．
外部オブジェクト	クラス名 オブジェクト名(コレクション名)	オブジェクト図において当該クラスと関連を持っているオブジェクトまたはそのオブジェクトによる間接参照オブジェクトである．関連を表すときは，子オブジェクトと同じく多重度が＊のときは，コレクション名が表示される．

表3　計算・宣言・イベントの表記法
Table 3　Notation for computation, declaration, event sending.

表記名	表記	表記の表す意味
計算	○	代入文や四則演算などオブジェクトの内部状態を変更する計算を表す．
宣言	□	メソッド内部で局所的に利用されるオブジェクトの宣言を表す．
外部入力イベント	self イベント名	当該メソッドを起動する外部からのイベントである．いわゆるメソッドの呼び出しを表している．イベントトレース図1枚ごとに，必ず1つだけ図の先頭行に記述する．
他へのイベント	イベント名	当該オブジェクトから他のオブジェクトへのイベント送信を表す．受信オブジェクトにおける外部入力イベントとなる．
	イベント名	当該オブジェクトから他のオブジェクトへのイベント送信を表すが，行き先オブジェクトが3段表記されていているとき，コレクションオブジェクトへのイベント送信を表す．
内部イベント	イベント名	セルフオブジェクトの内部のメソッドを起動する際に用いる．このイベントのみ弧が途中で曲がっており，セルフノードへ再び戻っていることを表わす．

4.1　送信先オブジェクトの表現

イベントトレース図においては，頂点を縦棒で記述し，当該クラスのインスタンスオブジェクト（これをセルフオブジェクトと呼ぶ）とこれがイベントを送る際の送り先となるすべてのオブジェクトを，**表2**のように，その種類に応じてシンボルを分けて表す．なお，OMTなどのイベントトレース図とは異なりSOMEのイベントトレース図では，イベントの送信を行うのはセルフオブジェクトのみに限定する．これは，SOMEではプログラム生成可能な詳細な設計図を作成するために単一のメソッド内での具体的な反応動作（この1つとして他のオブジェクトへのイベント送信が含まれる）を詳細に指定することを目的としているのに対して，OMTでは分析においてクラス間のイベントの送受信を分析することを目的としているからである．

オブジェクト図とイベントトレース図は，設計視点が違うだけであることから，クラスの集約要素オブジェクト（表2の2行目に示す子オブジェクト）は同じでなければならない．オブジェクト図で関連している送信先オブジェクトは（表2の5行目に示す）外部オブジェクトとして表す．また，当該メソッドの引数であるオブジェクトや，その反応を記述する際に必要となる局所オブジェクトは，表2の3，4行目のように記述する．なお，返値となるオブジェクトへのイベント送信は，そのオブジェクトのイベントトレース図中で記述すればよいので，当該メソッドのイベントトレース図中では図示しない．また，子オブジェクト，外部オブジェクトへの関連が，オブジェクト図においてその多重度が＊でありかつコレクション名が記述されているときは，これらオブジェクトのラベルは表2の2，5行目に示すようにこのコレクション名を含めた3段表記になる．

4.2　イベント送信の表記法

一般に，メソッドが行う反応動作は他オブジェクトとの相互作用の観点から，局所オブジェクトの宣言，属性やオブジェクトに対する計算，他オブジェクトへのイベント送信の3つに分けられる．これらを，**表3**に示すような別々のシンボルで，図フィールドに記述する．なお，コレクションオブジェクトに対するイベント送信と，コレクションの要素オブジェクトに対するイベント送信とで，表3の4行目に表すように矢印の先の形を分けて表現する．

4.3　制御の表記法

メソッド内で計算やイベント送信は単に時系列的に行われるのではなく，条件分岐（if文，switch文）や繰返し（while文，do-while文）による特定の制御条件の下で行われる．SOMEのイベントトレース図ではこの制御条件も，**図7**に示すようなシンボルを用いて表す．この図は，各制御構造の表記を記述した際の初期状態を示しており，図のシンボルがパレットから行にドラッグすると，それらのシンボルを自動的にセルフオブジェクトを表す縦棒のそばに表示され，さらに点線で示した行に対応するテキストフィールドが入

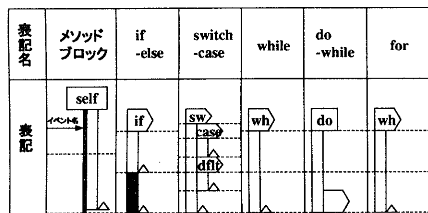


図7 イベントトレース図での制御論理の表記
Fig. 7 Notation for control logic in event-trace diagram.

力可能になる．制御の入れ子構造は，セルフオブジェクトを表す縦棒の右側に制御範囲の長さのブロックを層状に重ねて記述する．

結局，イベントトレース図の各行には，計算か宣言か制御を表す図形シンボルを 1 つ記入し，その具体的な内容をテキストフィールドに C++に準拠した形で記入する．また，これらがその内容の一部として他オブジェクトへのイベント送信を含めば，その送信先へイベントを表す弧を引く．なお，実際のこれらの描画においては，まず図形シンボルを選び，新規行にドラッグして，次にそのテキストを入力する．すると，SOME がその内容を構文解析し，イベント送信を含むなら，そのイベント名を名前に持つメソッドを持つオブジェクトをオブジェクト図から検索し，自動的にこのオブジェクトを表す縦棒へのイベント弧を描画する．もしそのようなメソッドを持つオブジェクトがなければ，エラーメッセージを log ファイルに出力する．

我々が制御を表すのに用いたこれらのシンボルは，従来から手続きの表現に用いられていた YAC 図[14]などの構造化フローチャートの表記法と同様である．ただ，我々のものは，構造化フローチャートの表記法にオブジェクト指向設計で重要なイベント送信の視覚化に関する表記法を C++の構文に準拠して体系化したものである．さらにコード生成に関しては，イベントの計算や制御条件がすべて C++の構文に従った方法で記述されている場合にはそれらがそのままメソッドの処理内容として展開されるようになっている．もしそれらが一般の文章であるならば，コメントとして生成されるようになっている．

5. ロジックテーブルと LOLA

5.1 プロセスクラスとファイル/コレクションクラス

ロジックテーブルと LOLA 言語を用いてビジネスプログラムのメソッドの処理記述を簡潔に行うためには，オブジェクト図において，問題を構成するクラスを，本質的な計算機能を表すプロセスクラス（例：図 5 の SalaryCalculation），データを格納している外部ファイルを表すファイルクラス（例：図 5 の MasterFile），と計算の途中で同一形式のデータを一時確保しておく入れ物を表すコレクションクラス（例：図 5 の NameRecord）に分けて構成する．このうち，その機能をロジックテーブルで実際に記述するのはプロセスクラスだけであり，他のクラスのメソッドは SOME が自動生成する．

5.2 高水準言語 LOLA

ビジネス処理などの特徴であるように，メソッド内の制御が主として条件分岐中心であるときは，ロジックテーブルでその機能を記述するのが，視覚的にも分かりやすい．さらにこの場合は，高水準言語 LOLA を利用できる．LOLA は，固有の条件式，処理文，条件値，実行指示記号を，ロジックテーブルに記入することで，メソッドの機能を簡潔に表現できる C++を基本としたビジュアルな高水準言語である．

給与計算クラスに対する 4 つのメソッドに対するロジックテーブルを**図 8** に示す．このように，ロジックテーブルは，6 つのエリア（図 8 の (3) に各エリアの番号を入れた）からなる表形式の設計図である．各エリアにはメソッドが行う処理について以下に示すような内容を記述する．

(1) 初期処理部：当該ロジックテーブルが呼ばれたときに最初に 1 回だけ実行する処理を記述する．

(2) 前処理部：繰返し処理を行うときに，本体処理を行う前に行う処理を記述する．ここに記述された処理は，ロジックテーブルが再実行されるたびに実行される．

(3) 宣言部：当該ロジックテーブルのみで有効となる局所変数を宣言する．

(4) 条件部：メソッドが実行中にとりうる状態を表す LOLA 条件式を記入する．LOLA 条件式には，**表 4** に示す 7 種類がある．ロジックテーブルが実行されるたびに条件式が評価され，特定の条件値を返す．複数個の条件式を条件部に記入することで，メソッドが遭遇するどんな実行状態も表現できるようになっている．実際，表 4 の第 7 行目の論理式 S は C++における条件式を表しており，したがって，これを用いればメソッド内の処理を分けるどんな条件も表現できる．これ以外の 5 つの条件式はファイル処理やコレクション処理を行うメソッドがよく遭遇する条件を簡潔に表現するために導入したものである．

(5) 処理部：メソッドが行う処理を LOLA 処理文を

(1) SalaryCalculate メソッド

```
LogicTableDiagram : SalaryCalculation::SalaryCalculate()
MakeNameMap();

MC(master;DepartmentID,MemberNo)                  | Y | Y | Y | N | E
MC(overtime;DepartmentID,MemberNo)                | Y | N | E | - | -

OvertimeCalculate();                              | X |   |   |   |
OvertimePay = 0;                                  |   | X | X |   |
Salary = master->BasicPay + OvertimePay;
newmaster->DepartmentID = master->DepartmentID;
newmaster->MemberNo = master->MemberNo;
newmaster->BasicPay = master->BasicPay;
newmaster->HourlyPay = master->HourlyPay;
newmaster->TotalPay = Salary + master->TotalPay;
fprintf(newmaster->_fp, "部No %2s 社員No %2d 基本給 %6ld 単価 %4d 合計 %6ld¥n",
    newmaster->DepartmentID,newmaster->MemberNo,newmaster->BasicPay,
    newmaster->HourlyPay,newmaster->TotalPay);
salarylist->DepartmentID = master->DepartmentID;
salarylist->MemberNo = master->MemberNo;
salarylist->Salary = Salary;
salarylist->MemberName = GetName(master->MemberNo);
fprintf(salarylist->_fp,"¥n部No %2s 社員No %2d 名前 %10s 給料 %6ld ",
    salarylist->DepartmentID,salarylist->MemberNo,
    salarylist->MemberName,salarylist->Salary);
Do DepartmentTotal();                             | X | X | X |   |
DoAgain;                                          | X | X | X | X |
```

(2) OvertimeCalculate メソッド

```
LogicTableDiagram : SalaryCalculation::OvertimeCalculate()

CB(overtime;DepartmentID,MemberNo)                | H | B | B | B | B | T | E
overtime->Category == {1,2,3}                     | - | 1 | 2 | 3 | 0 | - | -

OvertimePay = master->HourlyPay * overtime->Overtime;                   |   | X |   |   |   |   |
OvertimePay = long(float(master->HourlyPay * overtime->Overtime) * 1.5); |   |   | X |   |   |   |
OvertimePay = long(float(master->HourlyPay * overtime->Overtime) * 1.2); |   |   |   | X |   |   |
OvertimeTotal = 0;                                                      | X |   |   |   |   |   |
OvertimeTotal += OvertimePay;                                           |   | X | X | X |   |   |
DoAgain;                                                                | X | X | X | X | X |   |
```

(3) MakeNameMap メソッド

```
LogicTableDiagram : SalaryCalculation::MakeNameMap()
①
②                                                 ③
GT(namefile)                 ④                    | H | B | E
                                                    ⑥
NameRecord* namerecord = new NameRecord;
namerecord->MemberNo = namefile->MemberNo;
namerecord->MemberName = namefile->MemberName;   ⑤
NameMap[namefile->MemberNo] = namerecord;         |   | X |
DoAgain;                                          | X | X |
```

(4) DepartmentTotal メソッド

```
LogicTableDiagram : SalaryCalculation::DepartmentTotal()

LC(master;DepartmentID)                                          | F | I | U | L | E

salarylist->DepartmentTotal = 0;                                 | X |   | X |   |
salarylist->DepartmentTotal += Salary;                           | X | X | X | X |
fprintf(salarylist->_fp,"部合計 %6ld",salarylist->DepartmentTotal); |   |   | X | X |
```

図8　給与計算クラスの4つのメソッドに対するロジックテーブル
Fig. 8　Logic table for four methods of salary calculation class.

使って列挙する．LOLA 処理文には，**表5**に示す5種類がある．特に，DoAgain 命令はロジックテーブルで指定された処理を再実行する命令であり，メソッド内の繰返しを表現するのに用いる．表5の6行目のC++処理文はC++における任意の計算式であり，これによってC++で記述できる処理はロジックテーブルでも記述できるようになる．これら以外の4つの命令はロジックケーブルの初期処理部や前処理部の実行を制御するための命令である．

(6)　規則部：どのような条件のときにどのような処理を行うのかを記述する．条件部の各条件行に記入した条件式とその条件行内の個々の規則列の条件値欄に記入した条件値とのペアによって，メソッドの1処理状態を正確に表現する．各処理状態のときに実行すべき処理文が処理部にあれば，それが記述されている処理行とこの規則列が交差する実行指示子欄に実行指示記号Xを記入する．

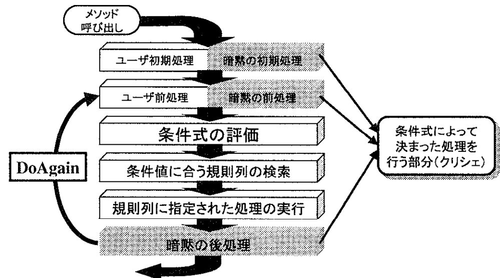


図9　ロジックテーブルにおける繰返し処理
Fig. 9　Repetition of processing in a logic table.

ロジックテーブルから生成されるプログラムの動作方式を**図9**に示す．この図に示すように，メソッドが呼び出されると，指定された初期処理と前処理が行われ，次に条件部に記入された条件式が評価され，現在のメソッドの状態を表す条件値が返される．次にこの条件値の組合せを持つ規則列が選ばれ，そこに実行指示された処理のみが実行される．この中に，DoAgain 命令があれば前処理の実行から再度繰り返す．このようにしてロジックケーブルに指定された処理が次々に行われ，DoAgain 命令を含まない規則が実行された段階でメソッドの処理は終了し，呼び出し側に戻る．

このようなロジックテーブル内での処理の繰返しにおいて，図9で灰色に網掛けされた部分は，条件部に記入された条件式の種類とその引数であるファイルとキーで構成されるデータ構造に依存して，ワーニエ法などの設計論を用いて一意的に導くことができる．したがって，これらに関するプログラムコードは条件式に依存した暗黙処理として，6章のプログラム生成で述べるように，SOME が自動生成する．したがって，利用者は白地で示される問題固有の条件や処理のみを記述すればよいので，処理記述が簡潔になる．

たとえば，図8の(1)の SalaryCalculate メソッドでは，master ファイルと overtime ファイルに対して，DepartmentID と MemberNo をキーとした照合処理を行っている．MC 式では両ファイルから現在読み込まれているオブジェクトのキーの値の小さい方を照合キー値として照合を行う．ここでは，master と overtime の両方に照合キー値と同じ値を持つオブ

表 4 LOLA 条件式
Table 4 LOLA condition expressions.

処理文名	機能
DoAgain	暗黙の後処理を行い、当該ロジックテーブルを再実行する。
Repeat	当該ロジックテーブルを再実行する。
SetNext	暗黙の後処理を行う。
Do	初期処理を伴わないロジックテーブルを実行する。
Init	ロジックテーブルの初期処理のみを実行する。
C++処理文	C++命令を実行する。

表 5 LOLA 処理文
Table 5 LOLA processing statements.

条件式	分類	取りうる条件値	意味
MC (ファイル名 F; キー名 K)	照合処理	Y、N、E、*	ファイルFから入力したオブジェクトのキー属性Kと現在の処理対象を表わす照合キー属性Kの値が等しい時 Yes,等しくないとき No。E は End of File の意味。 *はNまたはEの意味。
CB (ファイル名 F; キー名 K)	集計処理	H、B、E、T、*	当該メソッドが起動された直後の状態の時あるいは、ファイルFから入力したオブジェクトによって、キー属性Kが同一値の新しいオブジェクトグループに入った時 Head,グループ内での各オブジェクト処理状態の時 Body,現在のグループから抜け出ようとする終了状態の時 Tail。E は End of File の意味。 *はTまたはEの意味。
LC (ファイル名 F; キー名 K)		F、I、U、L、E、+、*	ファイルFから入力したオブジェクトが、キー属性Kが同一値のオブジェクトグループを構成する唯一のオブジェクトの時 Unique、グループの先頭オブジェクトの時 First、最終オブジェクトの時 Last、中間オブジェクトの時 Intermediate。E は End of File の意味。 *はLまたはEの意味。 +はLまたはUの意味。
GT (ファイル名 F)		H、B、E	当該メソッドが初期化された直後の状態の時 Head、各オブジェクトの処理状態の時 Body。E は End of File の意味。
ForEach (コレクション名 C ; C++論理式 L)	コレクション処理	H、B、T	コレクションCの中で条件式Lの成り立つ各オブジェクトに対して行う処理の実行状態を表す。当該メソッドが起動された直後の状態の時H、各要素処理状態の時B、処理対象となる全てのオブジェクトへの処理が終わった直後の状態の時T。
選択式 $S=\{S_1,S_2,\cdots,S_n\}$	分岐処理	0、1、…、n	式S=式Si となる最小の i≧1 が存在する時はその i、そのような i がない時 0。
論理式 S		Y、N	論理式Sの評価結果が真の時 Yes、そうでない時 No。

ジェクトが存在する場合（規則部第 1 列で YY の場合）には，OvertimeCalculate メソッドを呼び出し残業計算を行い，newmasterfile と salarylist に出力を行う．master のみにレコードが存在する場合には，残業計算を行わず，ファイルの出力のみを行う．最後の DoAgain を実行すると，DoAgain と条件部に記された 2 つの MC 式から生成される暗黙の後処理としてレコードが存在したファイルに対して次のレコードの読み込みを行い，さらに暗黙の前処理として，MC 条件式の再評価が行われる．また，図 8 の (2) の OvertimeCalculate メソッドでは，overtime ファイルに対して，DepartmentID と MemberNo をキーとした集計処理を行っている．すなわちこれらの 2 つの属性が同じオブジェクトを overtime ファイルから次々に読み込み，この Overtime 属性と master ファイル中の同一キー値を持つオブジェクトの HourlyPay 属性を掛けて，さらにこれを Category 属性が 1，2，3 かに応じて，1，1.5，1.2 倍して残業代を計算し，繰返しの最後でこれを OvertimeTotal 属性に足し込む．同一キー値を持つオブジェクトがつきると呼び出し側メソッドに戻ることを表している．

5.3 LOLA ファイルクラス

SOME には，ロジックテーブルでファイル処理を簡潔に表現できるようにするために CLTFile という特別のクラスが用意されている．LOLA 条件式の引数となるファイル F は，CLTFile を継承する必要がある．しかし，この継承は具体的には派生図において，クラス F へ CLTFile からの継承を表す弧を引くのではなく，クラス F のプロパティダイアログ（例：図 4 の NameFile）で指定する．このように指定されたクラスを LOLA ファイルクラスという．LOLA ファイルクラスには，ダイアログ指定において「InputFile」と「OutputFile」の 2 つの特性を選ぶことができる．LOLA ファイルクラス F に対して，これらの特性が選ばれていたら，クラス F にはそれぞれそのインスタンスオブジェクトの入力用および出力用のメソッドの実現部が生成されるようになっている．

したがって，クラス C のメソッド M がファイルを表すクラス F にアクセスしているとき，その機能をロジックテーブルと条件式を用いて簡潔に記述するには，以下の準備をする必要がある．

(1) クラス F が入力ファイルとして扱われるとき，必ず派生図でクラス F を「InputFile」特性を持つ LOLA ファイルクラスとして定義する必要がある．

(2) ファイル F のレコードの各項目は，クラス F の

属性として定義しなくてはならない．

(3) クラスＣはクラスＦと関連を持つ必要がある．

6. プログラム生成

C++プログラムは一般的に構造定義ファイル（.h）と動作定義ファイル（.cpp）から構成されている．これらがいかにしてSOMEの設計図から生成されるかを以下に述べる．

6.1 構造定義ファイル

構造定義ファイル（.h）では，クラスがどのような性質，機能，部品，関係を持つのかをそれぞれ属性，メソッド，集約，関連として宣言している．

6.1.1 クラス宣言の生成

派生図に登録されたクラスＣが入射弧を持つ場合は，その弧の始点頂点が表すクラスを用いて表１に示す意味に従って，クラスＣの宣言を行う．この弧が継承を表す場合は，派生の際のアクセス権がダイアログから入力されているので，それに従って，public, protected，privateの表記を加える．

この際，クラスの宣言順を次のように定める．

- 基本クラスは派生クラスよりも先に宣言する．
- クラスの構成要素として使われているクラスは使われているクラスよりも先に宣言する．

これは，C++での文法エラーを防ぐためであり，この宣言順序の作成にはトポロジカルソートを用い，先行関係が正しく成立するようにした．

6.1.2 集約・関連宣言の生成

集約・関連の情報はオブジェクト図から抽出する．たとえば，**図10**(1)に示すように，クラスsampleがクラスＡのオブジェクトをａという名で，またクラスＢのオブジェクトをｂという名で集約要素として持っており，図10(2)に示すように，オブジェクトａがオブジェクトｂと関連assocを持ち，そのときのｂの呼び名がroleであったとする．このような集約と関連は，**図11**に示すように宣言される．これで分かるように，集約関係はその親クラスに，関連関係は当該クラスにおいて定義することになる．

なお，多重度が１以外の集約や関連の場合は，Microsoft社のテンプレートクラスを用いて，**図12**に示すように宣言する．すなわち，多重度が具体的な数値で記述されている場合は配列を，＊の場合はリストを，＊でさらにキーが記述されているときはマップを用いて実装する．なお，多重度１以外の関連の場合も同様に生成する．

6.1.3 属性・メソッド宣言の生成

属性の名前，およびメソッドの名前，返値の型，引数並びは，図５で示したように，オブジェクト図のダイアログを用いて入力され，そのまま生成される．

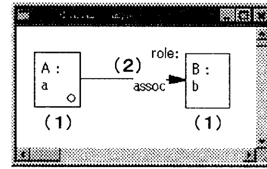
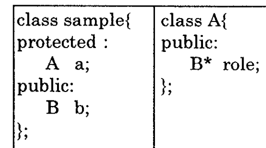


図10 集約・関連の図表現
Fig. 10 Expression of aggregation and association.



```
class sample{
protected :
    A  a;
public:
    B  b;
};
```

```
class A{
public:
    B*  role;
};
```

図11 集約・関連宣言の生成
Fig. 11 Coding of aggregation and association.

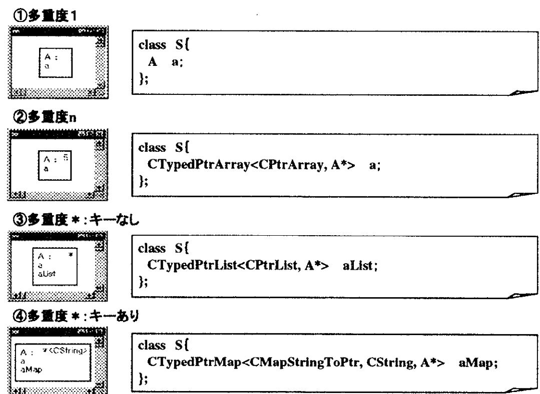


図12 集約表記と生成プログラム
Fig. 12 Coding of aggregation of multiplicity.

6.2 動作定義ファイル

動作定義ファイル（.cpp）には，メソッド（受信イベント）名，所属クラス名，メソッドの返値の型，メソッドの引数，受信イベントに対する反応動作（メソッド本体）の５つを記述する．このうち，メソッド名，クラス名，メソッドの返値，メソッドの仮引数に関しては前述したように構造定義ファイルにも生成されており，それと同様にして動作定義ファイルに生成する．一方，受信イベントに対する当該クラスの反応動作は，イベントトレース図かロジックテーブルに従って，以下の２つの項で論じるようにして生成する．

6.2.1 イベントトレース図からの生成

イベントトレース図のテキストフィールドの各行には，その行の図フィールドに記入された図形シンボルが表す計算，宣言，イベント送信の詳細を記述した，

制御文，条件式，計算式，コメントといったテキスト情報が記述されている．プログラム生成ボタンが押されれば，これらのテキストがそのままメソッド本体の記述として第1行から順に最後の行まで動作定義ファイルに出力される．

6.2.2 ロジックテーブルからの生成

[1] LOLA 基本クラスからの継承

ロジックテーブルから生成されるプログラムにはLOLA 言語による計算に共通した記述がかなりある．これらは以下に示すLOLA 基本クラス内に一括して生成する．問題ごとに生成されたプログラムを構成するクラスのいくつかは，LOLA 基本クラスを継承することで，これらの共通機能を再利用する．

(1) CLogicTable クラス：ロジックテーブル型メソッドの実行時に必要となる条件式の評価やキーの比較などの機能をもったクラス．ロジックテーブルで記述したメソッドを持つクラスはこのクラスを継承することになる．

(2) CLTFile クラス：LOLA 条件式の引数になるファイルに必要な機能やファイルを管理するための機能を持ったクラス．派生図でLOLA ファイルクラスとして定義されたクラスは，このクラスを継承することになる．

(3) CKeyCode クラス：キーの比較を行うメソッドなどを持つクラス．キーとしてLOLA 条件式に現れる属性があれば，CLogicTable や CLTFile の集約要素として宣言される．

次に給与計算問題の設計例から生成されたプログラムを構成するクラスを図13 に示す．この問題では，MasterFile, OvertimeFile, NameFile が入力タイプの LOLA ファイルクラスとして，NewMaster-File, SalaryList が出力タイプの LOLA ファイルクラスとして定義されている．また，SalaryCalculation クラスの4つのメソッドがロジックテーブルによって記述されているものとする．点線で囲まれた部分がSOME 設計図でユーザが設計した部分である．

[2] メソッドの展開法

SOME ではメソッドの機能記述にイベントトレース図とロジックテーブルの両方が混在する．したがって，生成されたメソッドが，この差を意識しないで相互に呼び出しあうことを可能にする必要がある．このために，ロジックテーブルで記述されてたメソッドに対して，図14 に示すような関数インタフェースを用意する．これら3つの関数を，ロジックテーブルに記述された内容にあわせて生成する．

_メソッド名_SetTable 関数は，規則列データの読み込みを行い，_メソッド名_Init 関数は，ロジックテーブルにおける初期処理を行う．_メソッド名_Body 関数はロジックテーブルの記述された条件評価と処理を行う．戻り値が "void" の場合は "return" が生成されないように展開することで，ユーザは通常の C++関数を呼ぶのとまったく同じ形式でロジックテーブル型メソッドを呼び出すことができる．

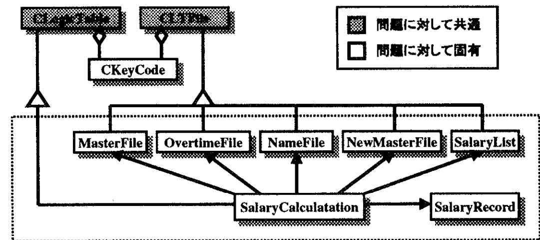


図13　給与問題の設計図から生成されたプログラムの構造
Fig. 13　The architecture of the generated program.

```
戻り値 クラス名::メソッド名(引数){
    _メソッド名_SetTable( );
    _メソッド名::Init( );
    return _メソッド名_Body(引数)
}
```

図14　メソッドの展開
Fig. 14　Generated code of method.

```
    void @クラス名@::_@メソッド名@_SetTable()
    {
*FOR J=1 TO 決定表の列数
        Map[{J-1}] = "@決定表文字列@";
*NEXT J
        Map[@決定表の列数@] = '\0';
    }
```

図15　クリシェの例
Fig. 15　An example of cliche.

[3] クリシェの設計と展開

我々は生成方式の開発の際に，図9 に示したような条件式に依存した暗黙処理などの決まりきったコーディングをクリシェという形で部品化した．クリシェには，生成プログラムの骨格になるフレームクリシェと条件式や処理文に対するコードを記述した部品クリシェがある．プログラム生成時には，フレームクリシェにオブジェクト図やロジックテーブルに記述された内容にあわせて部品クリシェの問題固有部分を編集して組み込む．クリシェを定義する際に図15 に示すような独自の表記法を用いた．ここでは，問題に依存しない共通な部分を C++言語で，問題に依存する部分をその内容の意味を表す@で始まる日本語で，またこれらのコードを繰り返し展開する制御条件などを*で始まる C++命令で表記する．実際のコード生成プログラム中では，これらのクリシェは展開制御条件に従っ

```
1  //DepartmentTotal 関数定義
2  void SalaryCalculation::DepartmentTotal()
3  {
4      _DepartmentTotal_SetTable();
5      _DepartmentTotal_Init();
6      _DepartmentTotal_Body();
7  }
8  //SetTable 関数定義
9  void SalaryCalculation::_DepartmentTotal_SetTable()
10 {
11     _DepartmentTotal_Map[0] = "F:XX " ;
12     _DepartmentTotal_Map[1] = "I: X ";
13     _DepartmentTotal_Map[2] = "U:XXX";
14     _DepartmentTotal_Map[3] = "L: XX";
15     _DepartmentTotal_Map[4] = "E:   ";
16     _DepartmentTotal_Map[5] = '\0';
17 }
18 //Init 関数定義
19 void SalaryCalculation::_DepartmentTotal_Init()
20 {
21     //初期処理部
22
23 }
24 //Body 関数定義
25 void SalaryCalculation::_DepartmentTotal_Body()
26 {
27     //宣言部
28
29     char _currCondVal[2] = "";
30     CString _Level1[_ALLKEY];
31     _Level1[0] = "DepartmentID";
32 _DOAGAIN:
33
34     //前処理部
35
36     //条件値の設定
37     _currCondVal[0] = _LC(master,_Level1);
38     _currCondVal[1] = '\0';
39
40     //条件に合う行の検索
41     int _rule =
42         _Search(_DepartmentTotal_Map,_currCondVal);
43
44     //処理部の実行
45     if (_DepartmentTotal_Map[_rule][2] == 'X') {
46         salarylist->DepartmentTotal = 0;
47     }
48     if (_DepartmentTotal_Map[_rule][3] == 'X') {
49         salarylist->DepartmentTotal += Salary;
50     }
51     if (_DepartmentTotal_Map[_rule][4] == 'X') {
52         fprintf(salarylist->_fp,"部合計 %6ld",
53             salarylist->DepartmentTotal);
54     }
55 }
```

図16 給与計算問題の設計例に対して生成されたプログラムの一部
Fig. 16 A part of the generated program from the design diagrams of the salary problem.

てクリシェに記述されたC++命令を書き出す fprintf 文群として表される．実際，図15に示したのは，規則列の読み込みを行うクリシェの例であり，これが図8の(4)のロジックテーブルに従って展開された結果が**図16**の8〜17行目のコードである．

次に，クリシェによるC++プログラムへの展開例をあげる．図16に示したのは図8の(4)に示した部ごとの合計を求め，部合計行を印字する DepartmentTotal メソッドに対して生成されたプログラムである．

```
void main (void) {
    SalaryCalculation* __project =
      new SalaryCalculation();
    __project->start();
    delete __project;
};
```

図17 main 関数の生成
Fig. 17 Generation of main function.

次に各部分についての説明をする．

4〜6 規則列の読み込み，初期処理の呼び出し，主処理の呼び出しを行う．ユーザがロジックテーブル型メソッドを呼び出したときには，実際にはこの部分が呼ばれることになる．

11〜16 条件値と実行指示子を配列として展開．

21〜22 初期処理部に記述された初期処理を行う．

27〜28 宣言部に記述された局所変数の宣言を行う．

34〜35 前処理部に記述された前処理を行う．

36〜38 LOLA 条件式ごとに用意されている専用の関数（この例では_LC）をファイル名やキー名を引数にして呼び出すことで，現在のメソッドの実行状態を表す条件値を求め，_currCondVal 配列に代入する．

40〜42 _currCondVal 配列に格納された条件値と同じ値を持つ文字列を_メソッド名_Map[i] 配列から探し，そのインデックス i を_rule に代入する．

44〜54 現在のメソッド状態を表す_メソッド名_Map[_rule] で与えられた規則の実行指示欄に実行指示記号 X があれば，それに対応する処理行に書かれた処理を実際に実行する．

6.2.3 main 関数の生成

生成プログラムの起動開始ポイントを示すために**図17**のような main 関数を生成する．この図に示すように，生成プログラムはプロジェクトクラスにオブジェクトを1つ生成し，これに start() イベントを送ることで生成プログラムの実行を開始する．したがって，利用者は，プログラムのスタート時に行いたい処理を “start” メソッド内に記述しておけばよい．

実際，事例の給与計算問題では start に「salary->SalaryCalculate();」を，イベントトレース図によって記述している．

7. 結　論

これまで述べてきたように，

- 構造化チャートで実証されたようなメソッド内の制御の視覚化を，イベントトレース図によって，オブジェクト指向モデリングに統合することができた．この統合のポイントは構造化チャートにメ

表 6 事例の設計結果
Table 6 Result of some problem design.

	クラス数	入力ファイル数	ETD数	LT数	使用した条件式	手作業による開発時間	LOLAによる開発時間
残業給与計算	6	2	2	3	MC,CB,LC,選択式	384	290
集計給与計算	6	1	2	1	CB	321	273
酒倉庫問題	14	4	25	12	ForEach,論理式	369	215
簡易住所録	4	1	6	3	GT,論理式	295	173
重複問題	6	2	0	6	MC,LC	392	253
分類問題	5	2	0	8	MC,選択式	405	240

ソッド呼び出しを表す横矢印棒線を導入して視覚化したことである．

- SPACE で実証したような決定表形式による視覚的な計算モデルの中にマクロな条件式による仕様の高水準化を，ロジックテーブルと LOLA 言語によって，オブジェクト指向モデリングに統合することができた．この統合にあたっては，ビジネス処理を表すクラスをプロセスクラスとファイル/コレクションクラスに分割したこと，コレクションクラスに対する処理状態を簡潔に表す ForEach 条件式を新たに導入したこと，およびコード生成時において継承機能を用いてクリシェ部品の数を減らしたことなどが本質的なことである．

以下にこの成果と今後の課題について論じる．

7.1 評価実験

我々はいくつかの事例を SOME で設計し，**表 6** に示すように，メソッドの機能記述をロジックテーブルによって行った．これらの設計図から生成されたプログラムはすべて正しく動作し，生成プログラムの正当性が確認された．この表に示すように，オブジェクト図を作成した後のプログラミング工程において，SOME/LOLA を使うことによって，制御ロジックを自分で考えテキストエディタで C++プログラムを作成するのに比べて，約 33%の効率化を達成できた．

実験の結果，以下のような定性的評価を得た．

- キーを用いたファイル処理やリスト処理に対して，相異なる処理状態を評価値で表す条件式を用いることで，C++よりも数段簡潔に表現できる．
- 決定表で処理論理を整理しているので，条件と処理の対応が視覚的に理解しやすい．よって，その後の修正も簡単に行うことができる．

これらから，LOLA は，視覚的にも処理の抽象化レベルにおいても，条件分岐中心のビジネス処理において，十分高水準な仕様記述言語であることが分かった．

7.2 評価と今後の課題

SOME におけるメソッドの機能記述については，まずイベントトレース図において，木構造チャート風のビジュアルな仕様を可能にし，さらにリストやマップによる多重度が * のオブジェクトの処理を簡易化できた．また汎用性に関しては，LOLA にはロジックテーブルの先頭への知的ジャンプである DoAgain 命令と他のロジックテーブルを呼び出す関数呼び出し命令があるので，順序，分岐，繰返しの 3 基本構造を表現できる．したがって，必要な条件と処理を表す適切な C++の式や文を記入すれば，C++で記述できるどんな問題も記述できるので，LOLA は十分汎用の仕様記述言語といえる．本研究成果を利用するにあたって，新たに習得すべき技術は表 5 に示す 5 つの処理文と表 4 に示す 7 つの条件式だけであり，これらは評価実験に参加した者によれば簡単に修得できることが分かった．したがって，ファイル処理を行う問題の作成においては，オブジェクト図の作成においてプロセスクラスとファイルクラスの存在を念頭に置いて設計すれば，その後のメソッドの論理設計を，むしろこれらのマクロ条件式を使うことによって，容易に行うことができ，コード生成の自動化の恩恵を受けることができる．実際，システムが完成後は表 6 に示す簡易住所録のように，筆者らの周りにおいて自発的に本システムを利用する者も出てきた．

一方，最近のアプリケーションはフレームワークやパッケージソフトを使って構築することが多い．この場合，Garlan が述べているように，システムは，

(1) カスタマイズされたパッケージからなる部分
(2) 一般的なプログラミング言語で記述されたアプリケーションの本体部分
(3) ウィンドウシステムなどのフレームワークから提供される部分

の 3 つのコード部分に分けられる[22]．SOME/LOLA はこのような環境下においても (2) の全体と (1) と (3) の一部の作成に用いることができる，実際 (2) の部分はその構成クラスを SOME で設計すれば，メソッド内の処理をロジックテーブルで記述できる．一方，(1) と (3) については，そのコードのほとんどはパッケージやインフラストラクチャから提供される．しかし，これらが提供する機能をアプリケーションにおいてどう使いこなすかを，それらの API コード（たとえば，ウィンドウに描画する命令）をアプリケーションの本質的処理コードとあわせてロジックテーブルに記述することで，SOME/LOLA を用いて記述できる．したがって，SOME/LOLA はビジネス処理以外の分野でも十分有用な高水準仕様記述言語であるといえる．

参 考 文 献

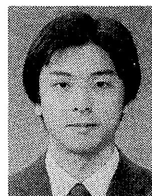
1) Ali, J. and Tanaka, J.: Automatic code generation from the omt-based dynamic model, *Proc. 2nd World Conference on Integrated, Design*

and Process Technology, Austin, Texas, Vol.1, pp.407–414 (1996).
2) Booch, G.: *Object-Oriented Design With Applications*, Benjamin/Cummings (1990).
3) Coad, P. and Yourdon, E.: *Object-Oriented Analysis*, Prentice Hall (1991).
4) Coleman, D., Arnold, P., et al.: *Object-Oriented Development The FUSION METHOD*, Prentice Hall (1994).
5) Goldberg, A. and Robson, D.: *Smalltalk-80: The Language amd its Implementation*, Addison-Wesley, Reading, Mass. (1983).
6) Harada, M. and Kunii, T.L.: A Design Process Formalization, *Proc. IEEE Computer Software & Application Conference, COMPSAC '79*, Chicago, pp.361–371 (1979).
7) 原田 実：COBOLプログラム自動生成システムSPACEにおける仕様の視覚化と抽象化，電子情報通信学会論文誌，Vol.J71-D, No.12, pp.2555–2562 (1988).
8) 原田 実（監修）：CASEのすべて，オーム社 (1991).
9) 原田 実，澤田隆史，藤沢照忠：構造化オブジェクトモデリング技法 SOMM とその構築環境 SOME，電子情報通信学会論文誌 (D-I), Vol.J79-D-I, No.3, pp.158–171 (1996).
10) Harada, M., Sawada, T. and Fujisawa, T.: A Structured Object Modeling Method SOMM and Its Environment SOME, *Systems and Computers in Japan*, Vol.27, No.11, pp.1–18, John Wiley & Sons (1996).
11) 原田 実，北本和宏，岩田隆志：制御構造とイベント送信を図示できる構造化オブジェクトモデリング環境SOME，情報処理学会 OO '97 シンポジウム論文集，pp.136–144 (1997).
12) 本位田真一，山城明宏：オブジェクト指向システム開発，日経 BP 出版センター (1993).
13) Jacobson, I.: *Object-Oriented Software Engineering*, Addison-Wesley (1992).
14) 加藤英雄：実践CASE入門—SDASによる事例，共立出版 (1990).
15) Parsons J. and Wand Y.: Using Objects for Systems Analysis, *Comm. ACM*, Vol.40, No.12 (1997).
16) Rumbaugh, J., Blaha, M., et al.: *Object-Oriented Modeling and Design*, Prentice Hall (1991).
17) Shlaer, S. and Meyer, S.J.: *Object-Oriented Systems Analysis*, Prentice Hall (1988).
18) Stroustrup, S.: *The C++ Programming Language*, Addison-Wesley (1986).
19) http://www.rational.co.jp/uml/uml.html
20) http:// www.rational.co.jp/products/rose/rose.html
21) Shlaer, S. and Mellor, S.: Recursive Design of an Application Independent Architecture, *IEEE Software*, Vol.14, No.1, pp.61–72 (1997).
22) Garlan, D., et al.: Architectural Mismatch: Why Reuse Is So Hard, *IEEE Software*, Vol.12, No.6, pp.17–26, IEEE Computer Society Press (1995).

(平成10年8月10日受付)
(平成11年5月 7 日採録)

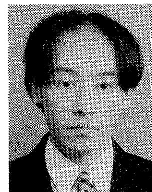
原田 実（正会員）

1951年生．1975年東京大学理学部物理学科卒業．1980年同大学理学系大学院博士課程修了．理学博士．同年（財）電力中央研究所担当研究員．1989年より青山学院大学理工学部経営工学科助教授．これまでに，再起グラフ理論の考案と応用研究，多くの自動プログラミングシステムの開発，オブジェクト指向分析の自動化システムの開発とその分析知識の自動更新の研究，アクティブメッセージの研究等を行う．1986年（財）電力中央研究所経済研究所所長賞．1992年人工知能学会全国大会優秀論文賞．訳書「ソフトウェアの構造化設計法」，編著書「自動プログラミングハンドブック」,「CASEのすべて」等．IEEE，ACM，AAAI，電子情報通信学会，人工知能学会，ソフトウェア科学会各会員．



水野 高宏

1998年青山学院大学理工学部経営工学科卒業．現在，同大学大学院修士課程在学中．



濱田 奨

1996年青山学院大学理工学部経営工学科卒業．1998年同大学大学院修士課程修了．同年富士ゼロックス（株）入社．