

ソフトウェア仕様化・設計の方法論の形式化について

佐伯元司

東京工業大学 工学部 電気電子工学科

本論文では、種々のソフトウェア仕様化・設計の方法論を形式的に扱うための、Entity-relationship modelと形式論理を用いた枠組について述べる。各方法論や仕様記述言語が提供しているソフトウェアのモデルより、種々の方法論/言語を表現できる共通のモデルを作り上げる。このモデルは、Event, Data, Processといった一般的なソフトウェアの要素を表す概念と概念間の関係によって構成される。このモデルを用いて実際の方法論/言語の分類を行なった。実際の方法論/言語のモデルは、これらの概念や概念間の関係に固有の制約を課して、共通モデルを特化したものとなっている。制約を $\forall\exists$ 型の述語論理式で記述し、その論理式の直観主義論理の枠組みでの存在証明によってその方法論の形式的な意味づけを与える。つまり、証明の結果得られた関数が方法論を関数的に記述したものとなっている。このような体系により、新しい方法論を導くことも可能になる。実例として、LOTOS言語による仕様化のための方法論の導出を行った。

Formalizing Software Specification and Design Methods

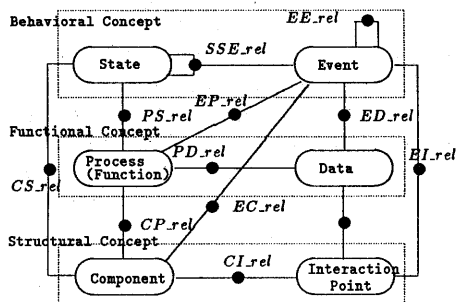
Motoshi Saeki

Department of Electrical and Electronic Engineering
Tokyo Institute of Technology

This paper presents a modeling technique for software specification and design methods such as Jackson Systems Development(JSD) etc. Our formalization is based on Entity-relationship model and on intuitionistic logic. We integrate the software models adopted by the various kinds of methods into a general model. This general model consists of the general concepts "Event", "Data", and "Process" etc. and of their relationships. It represents structure of the products through the specification and design process according to methods. Properties of the concepts and the relationships depends on the methods. We specify the properties by a set of $\forall\exists$ -form logical formula, which specialize the general model to the proper model of a method. If the formula are proved in an intuitionistic logic framework, a mathematical function can be extracted from the proof. This function produces something suitable for the properties of the concepts and their relationships in the method. So, we can consider that the proof strategy correspond to a part of the method because the method specifies the order of activities for making specifications. Furthermore we can derive new methods from the existing methods and/or the conceptual models of specification languages. In this paper, we illustrate the derivation of the method for LOTOS specification language.

1 はじめに

ソフトウェア開発の上流工程での開発効率を低下させる原因として、要求を完全に認識することができないこと、不適切な設計方法論やツールを使用してしまふことなどが挙げられる。これらの問題点を克服するために、ソフトウェアの仕様化/設計の方法論やそれに従った開発過程をモデル化し、何らかの形式的言語で記述する必要がある。この作業を行なうためには、仕様化/設計の方法論やそれに従った開発過程をモデル化し、何らかの形式的言語で記述する必要がある。この作業を行なうことにより、従来からの方法論や開発過程が本当に効率的なソフトウェア開発に役立っていたか、また役立っていないとすればどのようなものが望ましいかを議論することができる。さらに、このような方法論や開発過程のモデル化は、各方法論や開発過程に適したツールの作成や統合、開発履歴の蓄積やその再利用を行なうためにも不可欠である。本稿では、これまでに提案されてきた仕様化、設計の方法論やそれに従った開発過程を統一的な観点にたって整理し、形式的に扱うための1手法 [1] について述べる。



概念要素間関係

関係	関係している概念	関係の内容
EE.rel	Event, Event	現在の Event と次に起こる Event
SSE.rel	State, State, Event	現在の State にある Event が起こったときに遷移する State
PD.rel	Process, Data	Process の入出力 Data
CI.rel	Component, Interaction Point	Component が持っている Interaction Point
EP.rel	Event, Process	Process に起こる、あるいは Process が起こす Event
EC.rel	Event, Component	Component に起こる、あるいは Component が起こす Event
PS.rel	Process, State	Process が内部に持っている State
CS.rel	Component, State	Component が内部に持っている State
CP.rel	Component, Process	Component が持っている Process
EI.rel	Event, Interaction Point	Interaction Point で起こる Event
ED.rel	Event, Data	Event が携えている Data
ID.rel	Interaction Point, Data	Data が移動する際に使用する Interaction Point

図 1: システムを構成する概念要素とその間関係

2 システムの共通抽象モデル

仕様化/設計の方法論の最終目標は、対象システムの形式仕様の構築である。そのためにどのようにして現実のシステムを捉えるか、つま

りどのようにシステムをモデル化するか方法論にとって重要である。例えば、JSD 法 [2] ではシステムを通信しあう “Process” とみなしており、これは JSD 法特有の考え方である。システムのモデルを構成する概念要素 (例えば、JSD 法では “Process” など) は各方法論ごとに異なり、これらがどのような順序で抽出され、構成されていくかも方法論に大きく依存している。従って、モデルを構成する概念要素とそれらがどのようにして抽出されていくかによって、各方法論を特徴づけることができると思われる。統一的に種々の方法論を捉えるには、まずそれらに共通な概念要素を洗い出し、この共通な概念要素がどのような構造を持っているかを明らかにしなければならない。このようにして得られた共通の構造が方法論を形式化するためのモデルとなる。どのようにして共通モデルを作り上げるかについては、これまでにいくつかの研究がなされている [3,4,5,1]。本研究では、実際の方法論/言語やそれらを用いた開発過程 [6] を分析することによって、共通概念要素を抽出し、共通モデルを作り上げた。共通の概念要素とその間関係を ER-model で表現したものを図 1 に示す。共通モデルが持っている概念要素は、STATEMATE [7] や大規模のモデル [4] と同様に以下の 3 種類に大別できる。

1. Behavioral Concept: システムの時間的な振舞いをモデル化するための概念。State と Event
2. Functional Concept: システムの機能をモデル化するための概念。システムの機能単位を表す Process¹ とそれに対する入出力となる Data。
3. Structural Concept: システムの物理的な構造をモデル化するための概念。それ自身で存在可能な物理的要素である Component とそれらの間関係を表す Interaction Point。

これらは、同じ種類の概念要素同士だけではなく、異なる種類の要素とも関係を持っている。概念要素間関係の内容を図 1 の下部の表に示す。

複数の概念を合わせ持った概念も存在する。例えば、Data の概念を合わせ持った State の概念も存在し、これが Attribute の概念である。このような概念の重複は、方法論固有の現象なので、方法論ごとに考えることにする。また、方法論によっては、図 1 中のすべての概念要素や要素間関係を必要とするのではなく、一部のみにしか必要としないものもあったり、概念が統合されていたりしているものもある。例えば、LOTOS 言語 [8] には State 概念を表す言語要素は入っていない。概念間関係も、方法論によっては 1 対 1 対応になったり、1 対多対応になったり、その cardinality も種々である。このような概念や関係に対する方法論固有の制約は、次節以降で述べる。

3 実際の方法論/言語の分類

本節では、図 1 のモデルをもとに既存の方法論が支援する作業順序と生成物がどのようにになっているかを大まかに見てみよう。

3.1 方法論/言語が持っている概念との対応

表 1 に図 1 の共通モデルの概念要素と各方法論や仕様記述言語が提供しているモデルの概念要素との対応を、図 2 に共通モデルに沿って整理した各モデルの構造を示す。図 2 は、各方法論や言語によって作成、記述された最終生成物、つまりシステムの形式仕様の一般的構造も表している。図中からは、仕様作成に関して関連のない概念要素は除いた。例えば、PAISley [9] における Event 概念は、関数の評価開始と評価終了であるが、これらの概念は PAISley 実行系のみに必要な概念であって、仕様作成時には作成者は考慮する必要がない。そのため、PAISley の状態遷移を表す関係 SSE.rel は Event 概念の関与を必要としない。本稿では図 1 の 3 種類の Concept を持っていると思われる 7 つの方法論や言語を選択した。

¹従来の呼び方に従うのであれば、Function とすべきであるが、ここでは SA 法での呼び名 Process を使用した。

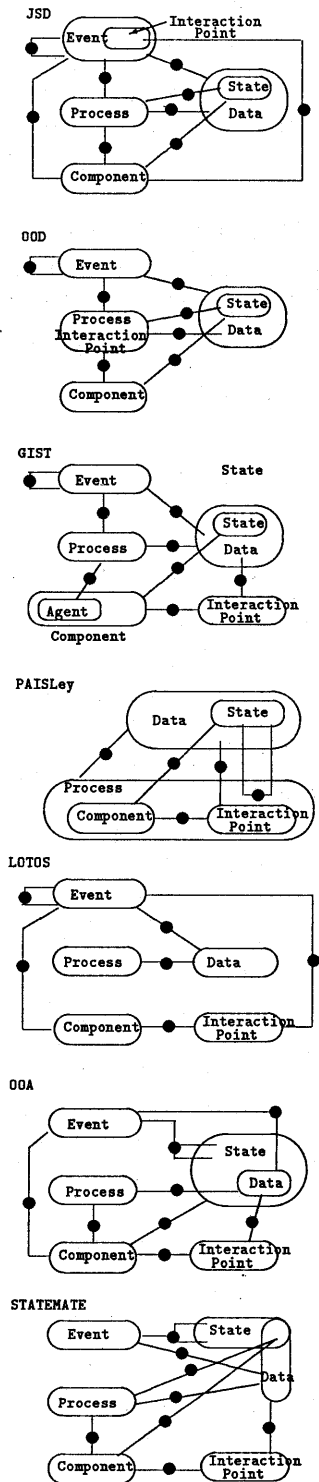


図 2: 各方法論/言語のモデル

表 1: 実際の方法論の概念要素との関係

	方法論の概念要素	共通モデルの概念要素
JSD[2]	[Modeling Stage] Entity Action Common Action [Network Stage] Model Process Function Process State Vector Data Stream	Component Event Interaction Point Process Process State Data
OOD[10]	Object Method Message Protocol Message Sending Instance Variable Class Variable	Component Process Interaction Point Event State, Data (Process の) State (Component の)
GIST[11] (ER-model)	Object, Agent Relationship Action Action の起動 (Daemon) Operation	Component Interaction Point Process Event Event
LOTOS[8]	Process Event Gate ADT の Selector ADT の Constructor	Component Event Interaction Point Process Data
PAISLey[9]	Process Channel Successor Mapping Successor Mapping の Data Mapping Mapping の評価開始 評価終了	Component Interaction Point SSE_rel State, Data Process Event
OOA[12]	Object Interaction Relationship State Attribute Process Data Store	Component EC_rel Interaction Point State State, Data Process State (Process の)

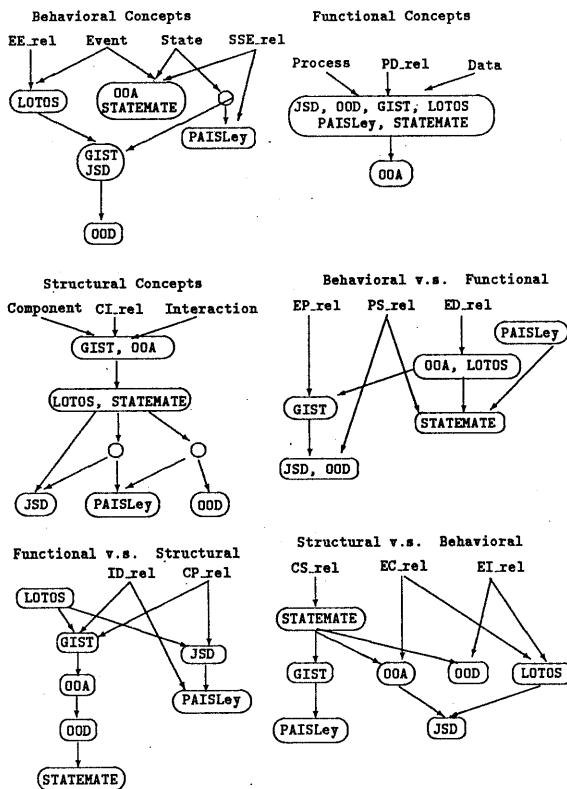


図3: 方法論/言語間の階層

3.2 生成物から見た分類

各方法論が最終目標とする生成物は図1のモデルを特化したもの、つまり図1のサブクラスとなっている。方法論Aに課せられている制約 C_A が方法論Bの制約 C_B よりも強い、つまり C_A ならば C_B が成り立つとき、AのモデルはBのサブクラスである。実際の方法論や言語が持っている概念、概念間の関係、それらに課せられている制約をもとに得られたクラス階層を図3に示す。この図は、3種類のConceptおよびそれらの間の関係ごとに整理していったものである。矢印の方向に従って、概念、概念間の関係、制約が継承されていくことを表している。例えば、Behavioral Conceptから見た階層図では、LOTOS言語はEvent概念とEvent間の関係 $EE.rel$ を持っており、他のBehavioral Conceptに関する概念や関係は持っていない。それに対してGISTやJSDは、LOTOSが持っている概念、関係やそれらに対する制約を継承しているだけでなく、新たにState概念を持っている。さらに、制約はLOTOSのそれよりも強いものとなっている。

3.2.1 Behavioralな観点から見た分類

図3のBehavioral Conceptから見た各方法論/言語の特徴を述べる。大きな特徴としては、Event、State概念のどれを持っているか、またシステムの振舞いを $EE.rel$ 、 $SSE.rel$ のどれによって規定しているかである。ほとんどの方法論/言語は、Event、State概念をともに持っているが、LOTOSはEventのみ、PAISleyはStateのみしか持っていない。システムの振舞いを規定するのは、Event系列を表す $EE.rel$ か状態遷移関係を表す $SSE.rel$ で、どちらか片方があればよい。Eventしか持っていないLOTOSは $EE.rel$ 、Stateしか持っていないPAISleyは $SSE.rel$ で振舞いを表現することになる。表2にこれらの特徴をまとめたものを示す。 $EE.rel$ 、 $SSE.rel$ の構成法もProcessごとに作っていく場合とComponentごとに作っていく場合との2通りがある。JSD

表2: 方法論/言語が持っている振舞いを表す概念

	Event oriented $EE.rel$	State oriented $SSE.rel$
Component ごと	JSD, LOTOS	OOA
Process ごと	OOD, GIST	PAISley, STATEMATE

表3: 方法論/言語が持っているState概念

	Component		Process	
	Behavior state	Data state	Behavior state	Data state
JSD	x	○	x	○
OOD	x	○	x	○
GIST	x	○	x	x
LOTOS	x	x	x	x
PAISley	x	○	x	x
OOA	○	○	x	○
STATEMATE	x	○	○	○

では、抽出したComponentで起こるEvent系列を作り上げ、Entity Structure Diagramを作成する。この作業は、Componentごとに起こるEvent系列を $EE.rel$ としてまとめたことになる。Componentごとに作られた $EE.rel$ においては、関係のあるEventは同一Componentで起こったもののみということになる。同様にComponentごとにまとめられた $SSE.rel$ については、遷移前のState、遷移後のStateは同じComponentが持っているStateであり、遷移を起こさせるEventもそのComponentが起こるEventだけである。

概念が縮退している例として、State概念がData概念に含まれているような方法論/言語が多い。State概念は大きく分けて、どこを実行しているかという振舞い自身に直接関係したもの(Behavior State)と、ObjectのAttributeやStatusなどのように現在までの実行履歴以外にDataとしても使用されるもの(Data State)とがある。また方法論/言語によっては、ComponentしかState概念を持っていなかったり、Processしか持っていなかったり、あるいは両方とも持っていたりする。表3に各方法論/言語が持っているState概念を表す。また、大部分の言語/方法論において、ComponentやProcessとStateとの関係は、1対多対応となっているが、PAISleyでは1対1対応となっている。State、Event概念をともに持っているEvent orientedな方法論/言語(JSD, OOD, GIST)のStateはすべてData stateである。これは、Stateによって振舞いを特に規定する必要がないため、Dataとしての役割が強調されたためである。

Functional Conceptとの関係、つまり $PS.rel$ 、 $EP.rel$ 、 $ED.rel$ の有無、および制約の強弱から見ると、以下のようなことがわかる。Event概念を持っていないPAISleyを除いて、 $ED.rel$ を持っている。これより、EventにはData伝搬の役割も持たせていることがわかる。

Structural Conceptとの関係で特徴的なのは、Component間のつながりを表すInteraction Pointである。LOTOSとOODは、Component間の関係はEventで表現するため、どのInteraction PointでEventが起こるかを表す $EI.rel$ が存在する。

3.2.2 Functionalな観点から見た分類

Functional Conceptのみから見ると、OOAのみがData概念に対して制約を受けている。OOAでは、Processへの入出力データは対応するComponentのData stateとなっていなければならない。つまり、Data概念がState概念に縮退している。

Structural Conceptとの関連で特徴的なのは、Process概念とComponent概念との関係 $CP.rel$ である。LOTOSにはComponentとProcessの関係がないが、それ以外の方法論/言語は、1つのComponentが複数のProcessを持っているもの、ComponentがProcessに含まれているものなどに大きく分けられる。前者はOOA, GIST, OOD, STATEMATE

表 4: Structural Concept の縮退

	JSD	OOD	PAISLey
Component	$\subseteq$ Process		$\subseteq$ Process
Interaction Point	$\subseteq$ Event	$=$ Process	$\subseteq$ Process

表 6: 概念間の階層構造

JSD	Component $\leadsto$ Event Process $\leadsto$ Event
OOD	Component $\leadsto$ Process $\leadsto$ Event
GIST	Component $\leadsto$ Process $\leadsto$ Event
LOTOS	Component $\leadsto$ Event $\leadsto$ Data, Process
PAISLey	Component $\leadsto$ Process
OOA	Component $\leadsto$ State $\leadsto$ Process
STATEMATE	Component $\leadsto$ Process $\leadsto$ State

ATEであり、これらはComponentごとにProcessをまとめて整理するという、Component-orientedな考えである。この中でさらに、Processを持たないComponentの存在を認めているもの(GIST)と認めていないもの(OOA, OOD, STATEMATE)に分けられる。STATEMATEのDataは必ずInteraction Pointを流れなければならないため、OODのそれよりも強い制約が課せられていることになる。ComponentがProcessに含まれているものはJSDやPAISLeyであり、これはProcess概念のほうが広いため、Process-orientedな考えとみなすことができる。

3.2.3 Structuralな観点から見た分類

GISTとOOAの生成物のStructuralな部分は、Entity-relationship modelである。Structural Conceptは他の種類のConceptに縮退している方法論/言語が多い。表4にそれらを挙げる。縮退が多いということは、Structural Conceptは生成物に含まれる他の種類のConceptの構造をも反映しており、それゆえ他のConcept、Concept間のRelationship抽出の手がかりにもなり得ることを示している。3.4節で述べるように、実際に多くの方法論がComponentの抽出作業から取りかかるようになっていく。

3.3 階層構造による分類

この節では、各方法論/言語が提供している階層構造の組み上げ方についての比較を行なう。具体的には、図1のモデルのどのConceptが下位のモデルのどのConceptにどのように分解されるかによって分析を行う。基本的には、同じ種類のConcept同士が結合する。例えば、Component概念は下位レベルの複数のComponentに分解されることがほとんどである。表5に各方法論/言語が持っている分解メカニズムの特徴をあげる。表の列は、分解の意味、つまり分解して得られた概念をどのように結合すればもとの概念になるかを表している。AND型、OR型は文献[7]での意味、つまりある概念を複数の下位概念のAND結合(直積)で表現するか、OR結合(直和)で表現するかを表している。 $\subseteq$ はスーパー/サブクラス階層の意味を表す。例えば、STATEMATEでは、State概念に関しては、2種類の分解方法があり、下位のState概念にAND分解すること、OR分解することも可能である。なお、Data概念に関しては、通常データ構造の構成法(直積型、直和型など)をほとんどの方法論/言語が持っているため、特徴的な $\subseteq$ 型しか表には載せなかった。共通モデルは1つの階層レベルでのモデルを表しているが、方法論/言語によっては異なる種類のConcept間にも、別の意味での階層性が存在する。例えば、OODでは1つのComponentがいくつかのProcessを所有し、これらをカプセル化している。これはComponent概念とProcess概念との間に、Componentを上位とする階層構造があると見ることができ、1つのレベル内での概念間の階層性をまとめると表6のようになる。

3.4 作業順序からの分類

生成物とならんで重要なことは、作業の順序である。つまり、どのような順序で図1の概念要素を抽出していくかである。概念要素間には互いに図1に示すような関係があるため、実際の作業では概念間の依存関係に沿った作業順序が望まれる。ある概念の抽出作業が終了した後、全くそれと関係を持っていない概念の抽出作業に取り掛かるのは上策とはいえない。関係のある概念の抽出作業に取り掛かったほうが、それらの間の関係も抽出でき、概念の抽出作業自身もやりやすいことが予想される。また、概念よりもその間の関係の抽出を先に行なうのは、人間にとってはやり難いであろう。つまり、概念AとBとの間に関係があるとき、先に概念Aを抽出してから、Aに対応づける形でBを抽出していく。このような順序で作業を進めていくことにより、関係も同時に抽出でき、前作業結果も直ちに活かすことができる。現実の方法論はこのような順序で作業が進められるものがほとんどであろう。

現実の方法論/言語についての概念の識別、抽出順序を調べると以下のようにになっている。

1. JSD

$$\begin{aligned} \text{Event} &\rightarrow \left\{ \begin{array}{l} EC_rel \rightarrow \text{Component} \\ ED_rel \rightarrow \text{Data} \end{array} \right\} \rightarrow EE_rel \rightarrow CI_rel \\ &\rightarrow \text{Interaction Point} \rightarrow CS_rel \rightarrow \text{State} \rightarrow CP_rel \\ &\rightarrow \text{Process} \rightarrow \left\{ \begin{array}{l} PD_rel \rightarrow \text{Data} \\ PS_rel \rightarrow \text{State} \end{array} \right\} \\ &\rightarrow EP_rel \rightarrow EE_rel, ED_rel \end{aligned}$$

2. OOD

$$\begin{aligned} \text{Component} &\rightarrow CP_rel(CI_rel) \rightarrow \text{Process}(\text{Interaction Point}) \\ &\rightarrow PD_rel(ID_rel) \rightarrow \text{Data} \\ &\rightarrow \left\{ \begin{array}{l} PS_rel, CS_rel \rightarrow \text{State} \\ EP_rel \rightarrow \text{Event} \rightarrow EE_rel, ED_rel \end{array} \right\} \end{aligned}$$

3. GIST

$$\begin{aligned} \text{Component} &\rightarrow \left\{ \begin{array}{l} CI_rel \rightarrow \text{Interaction Point} \rightarrow ID_rel \rightarrow \text{Data} \\ CS_rel \rightarrow \text{State} \end{array} \right\} \\ &\rightarrow CP_rel \rightarrow \text{Process} \rightarrow PD_rel \rightarrow \text{Data} \rightarrow EP_rel \rightarrow \text{Event} \\ &\rightarrow EE_rel, ED_rel \end{aligned}$$

4. PAISLey

$$\begin{aligned} \text{Component} &\rightarrow CI_rel \rightarrow \text{Interaction Point} \rightarrow ID_rel \\ &\rightarrow \text{Data} \rightarrow CS_rel \rightarrow \text{State} \rightarrow SSE_rel \rightarrow \text{Process} \rightarrow PD_rel \\ &\rightarrow \text{Data} \end{aligned}$$

5. OOA

$$\begin{aligned} \text{Component} &\rightarrow \left\{ \begin{array}{l} CI_rel \rightarrow \text{Interaction Point} \rightarrow ID_rel \rightarrow \text{Data} \\ CS_rel \rightarrow \text{State}(\text{Data State}) \end{array} \right\} \\ &\rightarrow \left\{ \begin{array}{l} CS_rel \rightarrow \text{State}(\text{Behavior State}) \\ EC_rel \rightarrow \text{Event} \end{array} \right\} \rightarrow SSE_rel \\ &\rightarrow ED_rel \rightarrow \text{Data} \rightarrow PD_rel \rightarrow \text{Process} \rightarrow CP_rel \end{aligned}$$

LOTOSとSTATEMATEは言語であり、特に方法論が指定されていないため、ここでは作業順序は取り上げなかった。それに対しPAISLeyでは、文献[9]に述べられている仕様化技法を取り上げた。図中の $\rightarrow$ は作業順序を表すのみで、概念要素、要素間関係といった生成物の依存関係を表しているのではない。例えば、JSD法では、まずEvent概念を識別・抽出する作業を行ってから、Eventに関係のある(EC_rel, ED_rel) ComponentやDataの抽出作業にはいる。中括弧で囲まれた部分は、特に順序のつかない作業で、並行して行われる作業である。2つ以上の生成物が同時に得られるときは、それらをコンマで区切ってならべた。

これら5つの方法論/言語の作業中の視点の変遷は、おおまかには表7のようにまとめられる。表7から、まず最初にシステムの振舞いを捉えるBehavior Orientedな方法論とシステムの構造に注目するStructure Orientedな方法論とに分類できる。Structure Orientedな手法では、システム構造(ComponentとInteraction Point)を抽出した後、抽出した要素(Component)が他の要素に対してどのような機能を持ってい

表 5: 方法論/言語の階層化のメカニズム

分解の種類	Structure Component	Function		Behavior State
		Process	Data	
OR 型の分解 AND 型の分解 ⊆ 型の階層	STATEMATE, LOTOS OOD, PAISLey, LOTOS OOD	GIST, OOA, PAISLey, STATEMATE, LOTOS	LOTOS	STATEMATE STATEMATE

表 7: 作業順序の概略

視点の変遷	方法論/言語
Behavior → Structure → Function	JSD
Structure → Function → Behavior	OOD, GIST
Structure → Behavior → Function	PAISLey, OOA

るかに注目する手法（OOD や GIST）と抽出した要素の内部の振舞いはどうなっているかに注目する手法（PAISLey や OOA）とに分けられる。

前節の階層構造を表わした表 6 と比べると、JSD 法では階層の下位の Event 概念から上位の Component 概念へと組み上げているためボトムアップ的な手法、他の OOD, GIST, PAISLey, OOA は階層の上位概念から下位へ視点を変えていくためトップダウン的な手法とみなせる。

4 方法論の形式化

前節では、個々の方法論/言語を生成物の構造と概念要素に関する作業順序という観点で分類を行ったが、本節では図 1 をベースにして方法論全体をどのように形式的に捉えるかについて考察してみよう。形式仕様を作成していくプロセスは、図 1 の各概念要素 (ER モデルでの Entity) と要素間の関係 (Relationship) を求め、制約を満足するようなモデルのインスタンスを求めていくプロセスと考えられる。方法論は求め方に制約を与えるものである。ここでは方法論を、最終目標としている生成物（形式仕様）の構造に関する制約と、その構造を求めるための戦略に関する制約とに分けて考える。それらの制約は、各方法論固有のものであり、人間の求める作業を定型化してくれる働きがある。しかし、実際のプロセスを唯一に制限するものではない。その意味で、ある共通の性質を持ったプロセスを集めた 1 つのクラスが 1 つの方法論と見なすことができる。

4.1 生成物の構造に関する制約

仕様化・設計作業の最終目標は各方法論固有の生成物に関する制約を表す論理式集合を満足するようなインスタンスを求めることにある。LOTOS の図 1 に課せられた制約は以下になる。

LOTOS1 必要な概念と関係

Event, Process, Data, Component, Interaction Point, EE_rel , PD_rel , CI_rel , ED_rel , EC_rel , EI_rel

LOTOS2 Event は必ず Component に所属し、Component は必ず Event を持つ

$\forall e \in Event \exists c \in Component (< e, c > \in EC_rel)$
 $\forall c \in Component \exists e \in Event (< e, c > \in EC_rel)$

LOTOS3 Interaction Point は必ず Component に所属し、Component は必ず Interaction Point を持つ。

$\forall i \in InteractionPoint \exists c \in Component (< c, i > \in CI_rel)$
 $\forall c \in Component \exists i \in InteractionPoint (< c, i > \in CI_rel)$

LOTOS4 Interaction Point には、必ずその Component と関係のある Event が起こる

$\forall i \in InteractionPoint \exists e \in Event (< e, i > \in EI_rel)$
 $\wedge < e, c > \in EC_rel \wedge < c, i > \in CI_rel)$

LOTOS5 因果関係 (EE_rel) にある Event は、同一 Component で起こる Event のみである

$\forall < e1, e2 > \in EE_rel \exists c \in Component$
 $(< e1, p > \in EC_rel \wedge < e2, p > \in EC_rel)$

LOTOS6 Data は必ず処理もしくはは生成される

$\forall d \in Data \exists p \in Process (< p, d > \in PD_rel)$

LOTOS7 Event は Data を伝搬する

$\forall e \in Event \exists d \in Data (< e, d > \in ED_rel)$

4.2 作業（求めるための戦略）に関する制約

求められる生成物（中間生成物も含む）が満足すべき論理式を直観主義論理の枠組で解釈し、存在証明を行い [13]、その証明が成功すれば、方法論が得られているはずである。つまり、生成物の制約式 (LOTOS2 ~ LOTOS7 など) は、

$\forall x \in X \exists y \in Y P(x, y)$

の構文をしており、この論理式を証明すれば、 $P(x, f(x))$ を満たす関数 $f: X \rightarrow Y$ を構成することができる。関数 f は概念要素 x をもとに y を求める作業を意味している。

どの生成物の制約式からどのようにして証明していくかという証明戦略に応じて種々の方法論が得られる。つまり証明の戦略が個々の方法論における作業の流れに該当すると考えられる。例えば、LOTOS2 の第 1 式は Event から Component を求めていく作業に、第 2 式は逆に Component から Event を求めていく作業に対応している。この LOTOS1 の第 1 式の存在証明、つまり LOTOS1 の第 1 式を満足するような Component の存在を証明することを最初に行うような戦略は、まず最初に Event を求め、次にその Event に関係のある Component を求める作業を持つ方法論を表していると考えられる。方法論中の各ステップでどの論理式の存在証明を行うかというサブゴールを与えることによって、求め方の制約を表現することができる。

実際には、LOTOS2~LOTOS7 のような生成物に関する論理式は、充足しないような解釈が存在するため、外から人間がある種の補助定理を与えてやらない限り証明不可能である。補助定理は、必ずしも構成的に関数を作り出すことができるという意味での証明が可能である必要はなく、直観主義論理の公理系と成果物に関する論理式に矛盾しないものであればよい。この人間が補助定理として与える論理式が、設計プロセス中で人間が行わなければならない作業となる。自動的に行われる部分は、機械による支援が可能な部分である。従って、

方法論=証明戦略、証明木の構成法/探索法
 設計プロセス=成功ノードに至るまでの 1 つの探索パス
 人間の作業= 外部から与える補助定理

と考えることができる。証明木を作っていく段階で、枝をこれ以上延ばせない箇所に対してどのような補助定理を導入すればよいかを調べることによって、人間がどのような作業をしなければならないかが抽出される。作業に関する制約、つまり証明戦略は

< 証明すべきサブゴールの集合、人間が与える補助定理の集合 >

の列で表すことができる。

このように考えることの利点としては、方法論と実プロセスを同じ形式的枠組でとらえることができるだけでなく、

1. 本質的にどのような人間の作業が必要かを Theoretical に捉え、人間の作業パターンの抽出、分類が行えること。

2. 証明木の大きさ、探索経路の長さといった客観的な尺度で方法論の評価、方法論中に含まれる作業の評価が行なえること。
3. 方法論を持っていない仕様記述言語についてどのような方法論が可能かを探ったり、既存の方法論や作業から新しい方法論を導いたりする、いわゆる方法論の合成の可能性があること

といったことが考えられる。ただし、本研究では方法論に従った実際の開発過程の支援を Prover で行なおうとしているのではない、

ある方法論が規定している作業の流れを

$$Task_1, Task_2, \dots, Task_i, \dots$$

$$Task_i = \langle SubGoals_i, Lemma_i \rangle$$

$SubGoals_i$: ステップ i におけるサブゴールの集合

$Lemma_i$: ステップ i で与える補助定理の集合

で表す。JSD 法の例では、 $Task_1, Task_2$ は各々 Entity-Action Step, Entity-Structure Step に該当する。

$Task_i$ において、本質的に人間がやらなければならない作業と機械で自動的に行える作業は以下のように区別される。

人間の作業

$$U_{j < i} (SubGoals_j \cup Lemma_j) \vdash Lemma_i$$

自動的に行える作業 (機械の作業)

$$(U_{j < i} (SubGoals_j \cup Lemma_j)) \cup Lemma_i \vdash SubGoals_i$$

どちらの作業もこれまで証明された式。つまりこれまでの作業で得られた中間的な生成物を用いて論理式を証明するパターンとなっている。

どのような $Lemma_i$ を導入しなければならないかは、 $SubGoals_i$ とステップ i までに証明された論理式に依存し、これらは方法論によって規定される。

4.3 LOTOS における作業の導出

前節の枠組みに従って、LOTOS による仕様作成作業を形式化し、分析してみよう。LOTOS は言語であり、仕様作成のための方法論は現在研究されているが [5]、まだ確立されたものはない。LOTOS 言語が要求する生成物の存在証明を行なっていくことによって、LOTOS のモデルと矛盾しない 1 つの方法論を作り出すことになる。生成物に課せられた制約は、LOTOS2 ~ LOTOS7 の各論理式によって、表現されており、これらの式の存在証明を行なっていく。1 つの作業の流れを図 4 に示す。図中の $Event, Process, Data, Component, InteractionPoint$ は各概念要素の集合を表し、 $EE_rel, PD_rel, CI_rel, ED_rel, EC_rel, EI_rel$ は 2 つ組を元とする集合 (2 項関係) を表す。集合の Constructor は、空集合を表す Constructor ϕ と、集合 A に元 e を 1 つ追加する $e.A$ である。 h で始まる関数は、 $Lemma$ から抽出した関数、つまり人間が行わなければならない作業を表す。また、 SET は概念要素集合の集合を、 $RELATION$ は 2 項関係の集合を、 $arg1, arg2$ は 2 つ組の第 1 要素、第 2 要素を取り出す関数を表す。作業は 9 つに分かれ、それらの各々は前節で述べたような定式化を行なうと、以下のようになる。

$$\begin{aligned} Task.1 &= \langle \{(1)\}, \{(1)\} \rangle & Task.2 &= \langle \{(2)\}, \{(2)\} \rangle \\ Task.3 &= \langle \{(4), (5)\}, \{(3)\} \rangle & Task.4 &= \langle \{(6)\}, \{(6)\} \rangle \\ Task.5 &= \langle \{(8)\}, \{(7)\} \rangle \\ Task.6 &= \langle \{(11), (13), (14), (15)\}, \{(9), (10)\} \rangle \\ Task.7 &= \langle \{(17)\}, \{(16)\} \rangle & Task.8 &= \langle \{(19)\}, \{(18)\} \rangle \\ Task.9 &= \langle \{(21)\}, \{(20)\} \rangle \end{aligned}$$

人間は、Component, Data を抽出 ($Task.1, Task.2$) した後、Component が持っている State の抽出 ($Task.3$) を行う。LOTOS 言語には State 概念が存在しないが、現実世界の Component が持っている State を抽出し、それをもとに LOTOS 言語の Event を抽出していかうとする方法である。抽出した State を Event とみなす ((9), (10) 式) ことによって作業 ($Task.6$) が進められていく。この作業によって得られる LOTOS 記述は、文献 [5] で述べられている State Oriented Style による記述である。このことは、LOTOS 言語の State Oriented Style の記述を支援する方法論の一つを合成したこと他に他ならない。

図 4 の論理式によく出現する構文のパターンおよび、それを基に作業のタイプを分類すると以下のようになる。

1. $\exists c \in Concept$
Concept を抽出する作業 (手がかりなし)
2. $\forall c1 \in Concept1 \exists c2 \in Concept2$
Concept1 をもとに Concept2 を抽出
3. $\forall c1 \in Concept1 \exists c2 \in Concept2 (c1 = c2)$
この作業は、恒等関数を表す。つまり Concept1 を Concept2 とみなす作業であり、ビューの切り替えを行っていることになる。

5 おわりに

本稿では、実際のソフトウェアの仕様化・設計の方法論/言語を統一的に扱う手法について述べた。今後は種々の方法論/言語への適用を行い、共通モデルの洗練を含めて、その評価を行う予定である。

謝辞

本研究は、情報処理振興事業協会のプロトタイプング技術の調査研究会 [6]、情報規格会 FDT-WG 小委員会 (主査: 二木厚吉氏) での活動、研究成果に負うところが大きい。御討論ならびに御指導して頂いた各委員の方々に感謝致します。

参考文献

- [1] 佐伯. 仕様化・設計の方法論形式化のための一考察, 第 2 回ソフトウェアプロセスワークショップ, 1990.
- [2] M.A. Jackson. *System Development*. Prentice-Hall, 1983.
- [3] 古宮 他. 仕様記述過程モデル化のための実験と分析. 情報処理学会ソフトウェア工学研究会, 69, 1989.
- [4] 大槻. ソフトウェア仕様記述モデルの形態学. 情報処理学会ソフトウェア工学研究会, 71, 1990.
- [5] Topic 6.1 — Modeling Techniques and their use in ODP. ISO/IEC JTC1/SC21 WG7 N109, 1989.
- [6] ソフトウェアプロトタイプング技術の調査研究ワーキング委員会による研究成果報告. 情報処理振興事業協会技術センター, 1990.
- [7] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. *Statemate: a working environment for the development of complex reactive systems*. In *Proc. of 10th ICSE*, pages 396-406, 1988.
- [8] *Information processing systems — Open Systems Interconnection — LOTOS — A formal description technique based on the temporal ordering of observational behaviour*. ISO 8807, 1989.
- [9] P. Zave. The operational versus the conventional approach to software development. *Commun. ACM*, 27(2):104-118, 1984.
- [10] G. Booch. Object-oriented development. *IEEE Trans. on Soft. Eng.*, 12(2):211-221, 1986.
- [11] R. Balzer, N.M. Goldman, and D.S. Wile. Operational specification as the basis for rapid prototyping. *ACM SIGSOFT Software Engineering Notes*, 7(5):3-16, 1982.
- [12] S. Shlaer and S.J. Mellor. An object-oriented approach to domain analysis. *ACM SIGSOFT Software Engineering Notes*, 14(5):66-77, 1989.
- [13] et al. Constable, R.L. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

最終ゴールとなる式

$\exists Event \exists Process \exists Data \exists Component \exists InteractionPoint \exists EE_rel \exists PD_rel \exists CI_rel \exists ED_rel \exists EC_rel \exists EI_rel$
 $(LOTOS2 \wedge LOTOS3 \wedge LOTOS4 \wedge LOTOS5 \wedge LOTOS6 \wedge LOTOS7)$

作業ステップ	作業
1. Component の抽出	$\exists Component \text{ --- (1)}$
2. Data の抽出	$\exists Data \text{ --- (2)}$
3. Component が持っている Interaction Point の抽出	$\exists InteractionPoint (Vc \in Component \exists i \in InteractionPoint) \text{ --- (3)}$ $\{(3)\} \vdash \exists CI_rel (Vc \in Component \exists i \in InteractionPoint (< c, i > \in CI_rel)) \text{ --- (4)(LOTOS3)}$ $\{(3)\} \vdash \forall i \in InteractionPoint \exists c \in Component \text{ --- (5)(LOTOS3)}$
4. Component が持っている State の抽出	$\exists State (Vc \in Component \exists s \in State) \text{ --- (6)}$
5. State から次の State, Event を抽出	$\exists Event (\forall s1 \in State \exists s2 \in State \exists e \in Event) \text{ --- (7)}$ $\{(6), (7)\} \vdash \exists SSE_rel (\forall s1 \in State \exists s2 \in State \exists e \in Event (< s1, s2, e > \in SSE_rel)) \text{ --- (8)}$
6. State を Event とみなし, Event 系列を作る	$\vdash \forall < s1, s2, e > \in SSE_rel \exists e1 \in Event (s1 = e1) \text{ --- (9)}$ $\vdash \forall < s1, s2, e > \in SSE_rel \exists e2 \in Event (s2 = e2) \text{ --- (10)}$ $\{(9), (10)\} \vdash \exists EE_rel (\forall < s1, s2, e > \in SSE_rel$ $\exists e1, e2 \in Event (< e1, e > \in EE_rel \wedge < e, e2 > \in EE_rel)) \text{ --- (11)}$ $\{(11)\} \vdash \forall < e1, e2 > \in EE_rel \exists c \in Component \text{ --- (12)}$ $\{(12)\} \vdash \exists EC_rel (\forall < e1, e2 > \in EE_rel$ $\exists c \in Component (< e1, c > \in EC_rel \wedge < e2, c > \in EC_rel)) \text{ --- (13)(LOTOS5)}$ $\{(7), (9), (10)\} \vdash \forall c \in Component \exists e \in Event (< e, c > \in EC_rel) \text{ --- (14)(LOTOS2)}$ $\{(7), (9), (10)\} \vdash \forall e \in Event \exists c \in Component (< e, c > \in EC_rel) \text{ --- (15)(LOTOS2)}$
7. Interaction Point で起こ る Event の特定	$\vdash \forall i \in InteractionPoint \exists e \in Event (< c, i > \in CI_rel \wedge < e, c > \in EC_rel) \text{ --- (16)}$ $\{(16)\} \vdash \exists EI_rel (\forall i \in InteractionPoint$ $\exists e \in Event (< e, i > \in EI_rel \wedge < e, c > \in EC_rel \wedge < c, i > \in CI_rel)) \text{ --- (17)(LOTOS4)}$
8. Data より Process の抽出	$\exists Process (Vd \in Data \exists p \in Process) \text{ --- (18)}$ $\{(18)\} \vdash \exists PD_rel (Vd \in Data \exists p \in Process (< p, d > \in PD_rel)) \text{ --- (19)(LOTOS6)}$
9. Event に付随する Data の抽出	$Ve \in Event \exists d \in Data \text{ --- (20)}$ $\{(20)\} \vdash \exists ED_rel (Ve \in Event \exists d \in Data (< e, d > \in ED_rel)) \text{ --- (21)(LOTOS7)}$

最終的に得られる関数: $ext_LOTOS()$

$= \langle EVENT, PROCESS, hDATA, hCOMPONENT, INTERACTIONPOINT, ext_EE, ext_PD, ext_CI, ext_ED, ext_EC, ext_EI \rangle$

式番号	得られる関数の一部 (15) 式まで)
	ただし $make_set(\phi)(f) = \phi$ $make_set(c.A)(f) = f(c).make_set(A)$ とする
(1)	$hCOMPONENT := SET$
(2)	$hDATA := SET$
(3)	$INTERACTIONPOINT := SET$ $INTERACTIONPOINT = make_set(hCOMPONENT)(hExt.inp)$ $hExt.inp : hCOMPONENT \mapsto INTERACTIONPOINT$
(4)	$ext_CI := SET$ $ext_CI = make_set(hCOMPONENT)(ext_CI1)$ $ext_CI1(c) = \langle c, hExt.inp(c) \rangle$
(5)	$ext_i.c : INTERACTIONPOINT \mapsto hCOMPONENT$ $ext_i.c(hExt.inp(c)) = c$
(6)	$STATE := SET$ $STATE = make_set(hCOMPONENT)(hExt.state)$ $hExt.state : hCOMPONENT \mapsto STATE$
(7)	$EVENT := SET$ $EVENT = Event1(STATE)$ $Event1(\phi) = \phi$ $Event1(hExt.state(c).S) = arg2(hExt.state.event(hExt.state(c))).Event1(S)$ $hExt.state.event : STATE \mapsto (STATE \times EVENT)$
(8)	$ext_SSE := RELATION$ $ext_SSE() = ext_SSE1(STATE)$ $ext_SSE1(\phi) = \phi$ $ext_SSE1(hExt.state(c).S)$ $= \langle hExt.state(c), arg1(hExt.state.event(hExt.state(c))), arg2(hExt.state.event(hExt.state(c))) \rangle \cdot ext_SSE1(S)$
(9)	$ext_event1 : STATE \mapsto (STATE \mapsto (EVENT \mapsto EVENT))$ $ext_event1(s1)(s2)(e) = s1$
(10)	$ext_event2 : STATE \mapsto (STATE \mapsto (EVENT \mapsto EVENT))$ $ext_event2(s1)(s2)(e) = s2$
(11)	$ext_EE := RELATION$ $ext_EE() = ext_EE1(SSE_rel)$ $ext_EE1(\phi) = \phi$ $ext_EE1(< s1, s2, e > .A) = \langle e, ext_event1(s1)(s2)(e) \rangle \cdot \langle ext_event1(s1)(s2)(e), e \rangle \cdot ext_EE1(A)$
(12)	$ext_comp : EE_rel \mapsto hCOMPONENT$ $ext_comp(< ext_event1(s1)(s2)(e), e >) = c$ $ext_comp(< e, ext_event2(s1)(s2)(e) >) = c$ ただし $s1 = hExt.state(c)$ $s2 = arg1(ext_state.event(c)(hExt.state(c)))$ $e = arg2(ext_state.event(c)(hExt.state(c)))$
(13)	$ext_EC := RELATION$ $ext_EC() = ext_EC1(EE_rel)$ $ext_EC1(\phi) = \phi$ $ext_EC1(< e1, e2 > .A) = \langle e2, ext_comp(< e1, e2 >) \rangle \cdot \langle e1, ext_comp(< e1, e2 >) \rangle \cdot ext_EC(A)$
(14)	$ext_c.e : hCOMPONENT \mapsto EVENT$ $ext_c.e(c) = arg2(hExt.state.event(hExt.state(c)))$
(15)	$ext_e.c : EVENT \mapsto hCOMPONENT$ $ext_e.c(arg2(hExt.state.event(hExt.state(c)))) = c$

図 4: LOTOS 言語による設計プロセスの定式化の例