博 士 論 文

オブジェクト指向方法論のための動的モデルの形式化の研究

指導教官　片山 卓也 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学

伊藤 恵

要 旨

本論文ではオブジェクト指向方法論の動的モデルの形式的仕様記述モデルとして ObTS を提案し、これに Statechart 式通信モデルに基づく操作的セマンティクスを与える。さらに ObTS に基づく仕様記述言語 ObCL を提案し、ObCL の記述支援およびシミュレーションのための環境を構築する。また、動的モデルにおける通信モデルの形式化としてイベント依存グラフを提案し、制限を加えたイベント依存グラフの Statechart 式通信モデルでの実現可能性を示す。さらにイベント依存グラフの上での等価性を定義した上で、Statechart 式通信モデルで実行可能なイベント依存グラフは状態遷移図に手を加えることで非同期通信モデルでも実行可能であり、逆に非同期通信モデルでも実行可能なイベント依存グラフは状態遷移図に手を加えることで Statechart 式通信モデルでも実行可能であることを示す。

目 次

図 目 次

表 目 次

第 1 章

導入

オブジェクト指向方法論において動的モデルの記述にリアクティブシステムの仕様記述モデル Statechart[1] が広く使われている [3, 4, 5, 6]。Statechart は状態遷移図に階層性と直交性を導入したモデルで、並行動作する部分間の通信はイベント通信で表される。Statechart は個々のオブジェクトの動作をわかりやすく記述できるが、それ自身にはオブジェクト指向の情報隠蔽の概念について考えられていないので対象システム全体を記述するには向いているとは言えない。そのためオブジェクト間の通信にはイベントトレース図などを用い、個々のオブジェクトの Statechart と全体のイベントトレース図で動的モデルを記述とすることが多い。

Statechart については多くの研究がなされ、何種類もの通信モデルが考えられてきたが [2]、その典型的なものの 1 つとして Statechart を使用した CASE ツール Statemate[7] 等で採用されている Statechart 式通信モデルがある。これは Statechart 全体の動作をステップに分けて考え、あるステップで発生したイベントはその次のステップの間のみ有効であり、それが使われたか否かに関わらず破棄されるというものである。この通信モデルでは、並行動作する各部分が 1 ステップで 1 回の遷移しか起こさないので、ステップに同期して動作することになり、現実世界の非同期的な通信とは対応がわかりにくい面があるが、状態遷移のトリガとしてイベントを AND,OR,NOT などで結合したイベント論理式を書くことができ、これによって複数のイベントを任意の順番で受け取って遷移するなどといったイベントの待ち合わせなどを簡単に記述することができる。また、イベントは必ず 1 ステップの間だけ有効であることから、イベントを一時的に蓄積するバッファを全体で 1 つだけ用意すればよく、非同期的通信モデルで必要になるようなオブジェクト毎のイベントキューは不要である。

本論文では個々のオブジェクトだけでなくシステム全体をオブジェクト指向の概念に基づいて記述可能な仕様記述モデル ObTS を提案し、オブジェクト間の通信モデルとして Statechart 式通信モデルを導入した操作的セマンティクスを与え、さらに ObTS のための記述言語 ObCL を提案し、ObCL の記述支援およびシミュレーションのための環境を紹介する。

また、オブジェクト間の動作の依存関係を記述するイベント依存グラフを提案し、状態遷移図による個々のオブジェクト内の動作仕様記述と、イベント依存グラフによるオブジェクト間のイベント通信の記述によってシステム全体の動作仕様を表現可能とする。制限を加えたイベント依存グラフで表される動作はStatechart 式通信モデルでプロトタイプ実行が可能であることを述べる。

さらに、イベント依存グラフの上で等価性を定義した上で、Statechart 式通信モデルで実行可能なイベント依存グラフは状態遷移図に手を加えることで非同期通信モデルでも実行可能であり、逆に非同期通信モデルでも実行可能なイベント依存グラフは状態遷移図に手を加えることで Statechart 式通信モデルでも実行可能であることを示す。

論文の構成は、第 2章で背景を述べた後、第 3章では準備としてオブジェクトシステムにおける通信モデルとして Statechart 式通信モデルおよび非同期通信モデルの 1 つである個別実行モデルについて述べる。

論文前半の第 4〜7章は ObTS に関する部分で、第 4章でオブジェクト指向方法論のための形式的動作記述モデル ObTS を提案し、形式的なシンタックスを述べ、操作的セマンティクスを与える。第 5章では ObTS に基づく仕様記述言語 ObCL を導入し、ObCL の記述支援環境およびシミュレーション環境を紹介する。第 6章では ObTS および ObCL の記述例を示す。第 7章では ObTS および ObCL を他の記述モデルと比較する。

論文後半の第 8〜11章はイベント依存グラフに関する部分で、第 8章でイベント依存グラフを提案し、第 9章で制限されたイベント依存グラフが Statechart 式通信モデルで実行可能であることを述べる。第 10章でイベント依存グラフ間の等価性の定義を述べ、第 11章で Statechart 式通信モデルと個別実行モデルとの間でイベント依存グラフ上の等価性に基づく相互シミュレーションについて述べる。

最後はまとめと今後の課題である。

第 2 章

背景

2.1 オブジェクト指向方法論

オブジェクト指向開発/設計方法論 (以下オブジェクト指向方法論と略す) では実世界の概念をオブジェクトの集まりとしてモデル化する。個々のオブジェクトは構造と操作を持つが、その詳細はオブジェクト内に隠蔽され、外部にはデータ入出力や操作の呼び出しのインタフェース部分のみが見える。この情報隠蔽の概念により、仕様の変更に対しても変更の波及が一つもしくは一部のオブジェクトだけに留まることが多く、オブジェクト指向方法論に基づく設計/開発は仕様の変更に容易に対応できるという特徴がある。

2.2 Statechart

Statechart はリアクティブシステムの仕様記述やオブジェクト指向方法論の動作記述に使われている状態遷移図に基づくモデルで、状態遷移図に対して階層性を導入することで状態数や遷移数の爆発を抑え、直交性を導入することで並行動作の表現を可能にしたモデルであり、並行動作する部分間の通信にイベントのブロードキャスト通信が用いられる。

Statechart では個々のオブジェクトの動作はわかりやすく記述できるものの、大域変数によってのみデータが表されることや、このモデル自身は情報隠蔽などのオブジェクト指向の概念について考えられていないことにより、オブジェクトの集まりで表される対象システム全体の記述には向いているとは言えない

2.3 Statechart 式通信モデル

Statechart についてはさまざまな研究がなされており、複数の通信モデルが提案されているが、ここでは CASE ツール Statemate[7] で採用されているものを Statechart 式通信モデルと呼ぶこととする。

Statechart 式通信モデルでは全体の動作をステップという単位でわけて考え、個々のステップの動作の列が全体の動作になっている。各ステップにおいて並行動作する各部分はたかだか 1 回の状態遷移を同時

に行い、そこで出力されたイベントはすべて次のステップに状態遷移図全体にブロードキャストされる。このような通信モデルを使うことで状態遷移のトリガとしてイベントの論理式を書くことができ、複数イベントの待ち合わせなどが簡単に記述できる。また、イベントは 1 ステップのみ有効でブロードキャストのみであることから、並行動作する各部分が個別にイベントキューなどを持つ必要が無く、システム全体で単一のイベントバッファだけでイベント配送を表すことができる。

第 3 章

通信モデル

本題に入る前にオブジェクトの集合から構成されるオブジェクトシステムの通信モデルとして、Statechart 式通信モデルおよび個別実行モデルの 2 つの通信モデルについて述べておく。前者は Statechart で並行動作する部分間の通信に用いられているモデルをオブジェクト間の通信に利用したもので、後者は非同期通信に基づくものである。

3.1 Statechart 式通信モデル

Statechart 式通信モデルでは各オブジェクトはステップに同期して動作する。ステップとは

1. 前のステップで発生したイベントを全オブジェクトにブロードキャスト
2. 各オブジェクトを同時に動作 (各オブジェクトは最大 1 回状態遷移)
3. 出力されたイベントを収集

という動作であり、これを繰り返すことでオブジェクトで構成されるシステム全体を動作させていく。各

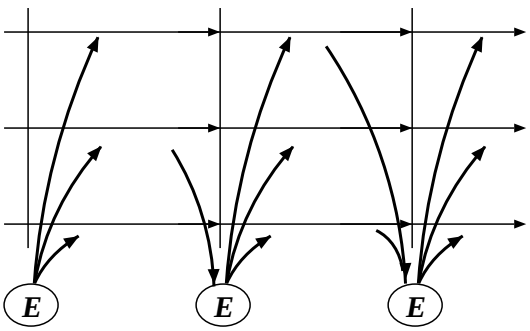

Figure 3.1: Statechart 式通信モデル

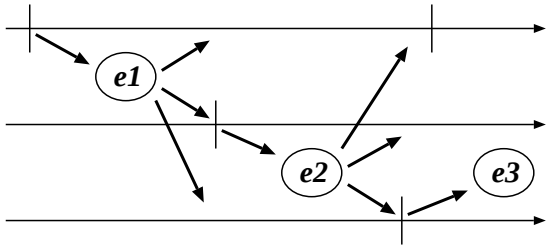

Figure 3.2: 個別実行モデル

オブジェクトを同時に動作させることから、この通信モデルでは状態遷移の入出力は単一のイベントでなくイベント集合となる。

各オブジェクトの状態遷移は瞬時に行われ、あるステップで発生したイベントは 1 ステップ後にはすべてのオブジェクトに到達する。到達したイベントは即座に遷移を引き起こすか、破棄される。

この通信モデルの元での各オブジェクトの状態遷移は

$$s_1 \rightarrow s_2; E_1[c]/E_2[act]$$

のような形で書くこととし、s_1, s_2が遷移元, 遷移先の状態、E_1, E_2は入力と出力のイベント集合である。c は遷移が実行されるための (入力イベント以外の) 条件で、act は遷移の際に実行される動作である。具体的な条件や動作は記述モデルによって異なるが、c の省略値は $true$、act が省略された場合は遷移以外の動作はしないことを意味する。状態遷移図上では s_1から s_2への矢印のラベルとして $E_1[c]/E_2[act]$ の部分を記述する。

あるオブジェクトのある状態を遷移元とするような状態遷移が複数あった場合、どのような入力が与えられても複数の遷移が同時に遷移可能となるように記述されることはないものとする。この条件の元で Statechart 式通信モデルは決定的に動作する通信モデルである。

3.2 個別実行モデル

個別実行モデルでは各オブジェクトは受け取ったイベントで個別に動作して、出力イベントをブロードキャストする。ただし、システム全体で複数の状態遷移が同時に起こることはないものとする。

この通信モデルでは状態遷移の入出力は単一のイベントで、発生したイベントは 0 より大きい有限の時間で自分および他のオブジェクトに到達する。状態遷移は瞬時に起こるものとする。

この通信モデルの元での各オブジェクトの状態遷移は

$$s_1 \rightarrow s_2; e_1[c]/e_2[act]$$

のような形で書くこととする。入出力が単一のイベント e_1, e_2になっていること以外は Statechart 式通信モデルの場合と同じである。

オブジェクト間のイベント到達時間については有限であるということ以外は規定されていないので、その与え方によってシステム全体として複数の動作が考えられる。

3.3 イベントシステム

オブジェクト間のイベント配送のメカニズムをイベントシステムと呼ぶこととする。イベントシステムには通信モデルが含まれるほか、オブジェクト集合を決定的に動作させるための情報– 例えばオブジェクト間の優先度やイベント伝達時間など – が含まれる。

Statechart 式通信モデルは通信モデル自体が決定的であるため、Statechart 式通信モデルのイベントシステムは唯一であり、これを E_{scm}とする。また個別実行モデルはイベント伝達時間等を指定することで複数のイベントシステムがあり得るので、その集合を E_{ind}とする。

第 4 章

ObTS

本章ではオブジェクト指向方法論のための動的モデルの形式的仕様記述モデル ObTS を提案し、形式的なシンタックスを述べ、操作的セマンティクスを与える。

4.1 概説

ObTS は記述対象のシステムの構造をオブジェクトの階層構造で、システムの動作を各オブジェクトの状態遷移/関数的な属性計算/属性付きイベントのブロードキャスト通信でモデル化したものである。

ObTS ではすべてのオブジェクトを最初から記述してあって、実行中に新しいオブジェクトを生成したり、存在していたオブジェクトが消滅したりすることはないこととする。また、記述対象のシステムの外部の環境も含めて ObTS で記述し、原則的に外部からのイベントは考えないものとする。階層構造の根にあるオブジェクトが動作し始めることで ObTS の動作が始まる。

4.1.1 オブジェクトの構成

オブジェクトは属性と状態遷移図を持っている。オブジェクトは動作の委譲のため内部オブジェクトを持つことが出来る。内部オブジェクト自身もオブジェクトであって同じ構造を持ち、階層的に定義される。また、複数の内部オブジェクトを並行動作するものとして記述できる。オブジェクトが起動されたときに最初に実行される状態遷移として初期遷移を定義しておく。

例 4.1 図 4.1のようにオブジェクトは角の丸い四角、状態は円、状態遷移は矢印、初期遷移は黒丸からの矢印、属性は四角で記述する。ここでは構造にのみ着目しているので、わかりやすいように本来状態遷移に付加される遷移規則のラベルは省略している。

この図ではオブジェクト X は属性 a_1, a_2、状態 s_1, s_2、内部オブジェクト Y から成っており、オブジェクト Y は属性 a_3, a_4、状態 s_3, s_4から成っている。

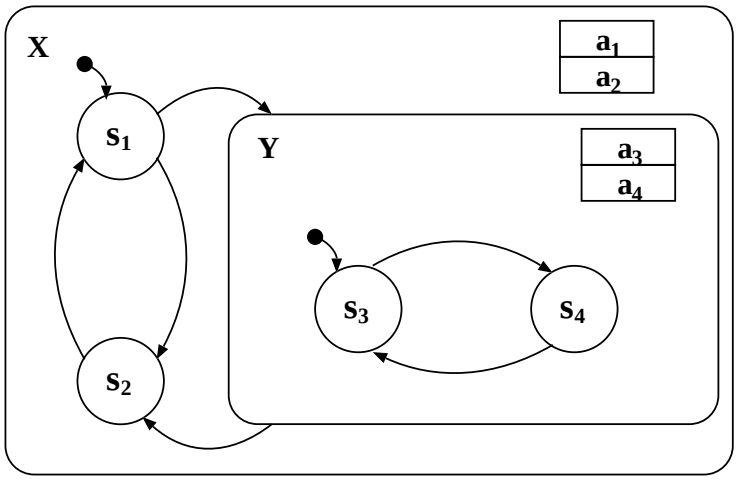

Figure 4.1: オブジェクトの構成

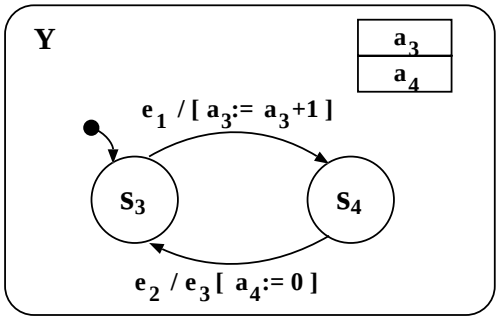

Figure 4.2: オブジェクトの動作

データ計算は関数的な属性計算によって行なわれる。オブジェクトの持つ属性はそのオブジェクトの局所的データであり、そのオブジェクトが状態遷移をするときのみその値が計算される。イベントの持つ属性は通信データであり、そのイベントを発生する状態遷移によって属性値が計算される。

状態遷移で参照できる属性は根のオブジェクトから遷移するオブジェクトまでのすべてのオブジェクトの属性と入力イベントの属性で、状態遷移で変化する属性は遷移するオブジェクトの属性と出力オブジェクトの属性である。

例 4.2 先程の図 4.1では、オブジェクト Y が遷移のときに参照できる属性は a_1, a_2, a_3, a_4と入力イベントの属性であり、遷移のときに変化する属性は a_3, a_4と出力イベントの属性である。

4.1.2 オブジェクトの動作

オブジェクトは入力イベントを受け取って状態遷移を行ない属性評価をして出力イベントを出す。

例 4.3 図 4.2は、先程の図 4.1の内側にあったオブジェクト Y の状態遷移に遷移規則のラベルを省略せず

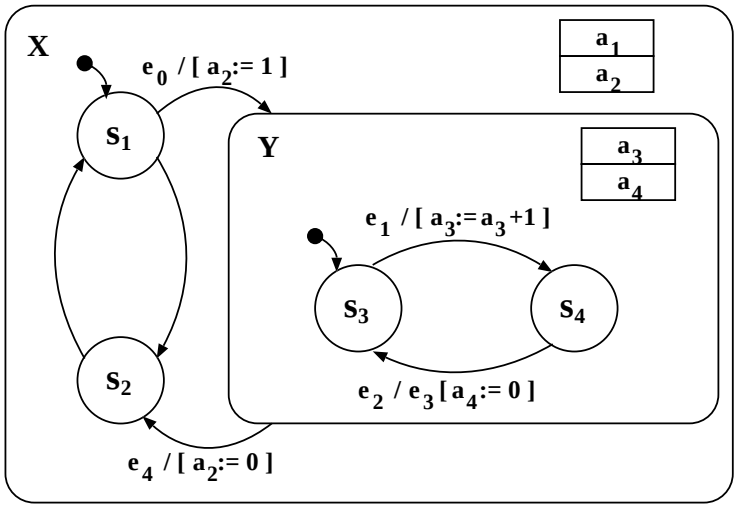

Figure 4.3: 内部オブジェクトの起動と終了

に付加したものである。

この例では、オブジェクト Y が状態 s_3であるときにイベント e_1を受け取れば状態 s_4への遷移が起こり、このとき属性 a_3の値が 1 増やされる。また、オブジェクト Y が状態 s_4であるときにイベント e_2を受け取れば状態 s_3への遷移が起こり、このときイベント e_3が出力され、属性 a_4の値が 0 になる。

内部オブジェクトへの遷移によって内部オブジェクトは起動され、内部オブジェクトからの遷移によって内部オブジェクトの動作は終了する。オブジェクトが起動されたとき、初期遷移として記述された状態遷移が実行される。初期遷移によってそのオブジェクトの最初の状態や属性の初期値を決めることができる。状態遷移の中で入力イベントなしに実行できるのは初期遷移だけである。

例 4.4 図 4.3は、図 4.1に対してオブジェクト X がオブジェクト Y を起動する遷移と終了させる遷移のラベルを追加したものである。

この例では、オブジェクト X が状態 s_1のときイベント e_0を受け取れば内部オブジェクト Y を起動し、X 自身は Y を実行中であるという状態になり、このとき属性 a_2の値を 1 にする。また、オブジェクト X が内部オブジェクト Y を実行中のときイベント e_4を受け取れば Y の実行を終了させて状態 s_2に遷移し、属性 a_2の値を 0 にする。

並行動作する内部オブジェクトは同時に起動され、同時に終了する。並行動作するオブジェクト間の通信は属性付きイベントのブロードキャスト通信によって行なう。

例 4.5 図 4.4はオブジェクト W の内部オブジェクトとして並行動作する 2 つのオブジェクト Z_1, Z_2があるような例である。ただし、Z_1, Z_2の状態名やほとんどの遷移ラベルは省略してある。

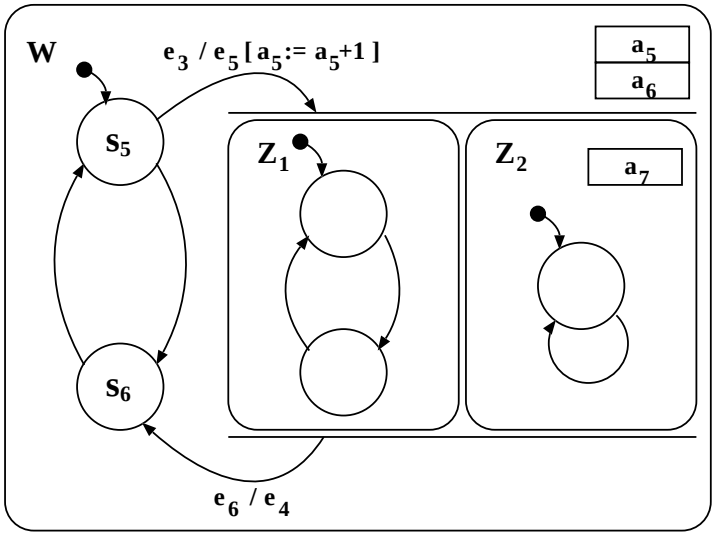

Figure 4.4: 並行動作の開始と終了

ここでは、オブジェクト W が状態 s_5であるときイベント e_3を受け取ると内部オブジェクト Z_1, Z_2を同時に起動し、このときイベント e_5を出力し、属性 a_5の値を 1 増やす。また、オブジェクト W が内部オブジェクト Z_1, Z_2を実行中にイベント e_6を受け取ると内部オブジェクト Z_1, Z_2を同時に終了させて状態 s_6に遷移し、このときイベント e_4を出力する。

4.2 シンタックス

本節ではObTS モデルの形式的定義を与える。

4.2.1 属性に関する定義

属性はオブジェクトとイベントに付加されている。A を属性の集合、D_Aを属性値のドメイン集合、op を D_A上の k 項演算子としたとき、属性式の集合 $AExp(A)$ は次のように定義される。

定義 4.1 $AExp(A)$

1. $n \in D_A$ならば、$n \in AExp(A)$
2. $att \in A$ ならば、$att \in AExp(A)$
3. $a_1, a_2, \cdots, a_k \in AExp(A)$ ならば、$op(a_1, a_2, \cdots, a_k) \in AExp(A)$

A を属性の集合、D_Aを属性値のドメイン集合、R を D_A上の k 項関係演算子としたとき、属性論理式の集合 $BExp(A)$ は次のように定義される。

定義 4.2 $BExp(A)$

1. $\mathrm{true}, \mathrm{false} \in BExp(A)$
2. $a_1, a_2, \cdots, a_k \in AExp(A)$ ならば $R(a_1, a_2, \cdots, a_k) \in BExp(A)$
3. $b_1, b_2 \in BExp(A)$ ならば $b_1 \wedge b_2, b_1 \vee b_2, \neg b_1 \in BExp(A)$

a を属性としたとき、ObTS 実行中の a の値を $v(a)$ と書くこととする。A を属性の集合としたとき、ObTS 実行中において A の属性値集合 $Av(A)$ は次のように定義される

定義 4.3 $Av(A) = \{(a, v) | a \in A, v = v(a)\}$

$(a, v) \in Av(A)$ のとき $Av(A)$ 中の属性 a の値を val に置き換えたものを $Av(A)[val/a]$ と書くこととする。ObTS 実行中において, 属性式 $exp \in AExp(A)$ と属性値集合 $Av(A)$ が与えられたとき、exp 中で使われている属性の値はすべて $Av(A)$ 中に与えられているので exp の値を評価することができる。その評価した値を $Val(Av(A), exp)$ と書くことする。

D_Aを属性値のドメイン集合、op は D_A上の k 項演算子としたとき

定義 4.4 $Val(Av, exp)$

1. $Val(Av(A), n) = n$ if $n \in D_A$
2. $Val(Av(A), a) = v$ if $(a, v) \in Av(A)$

3. $Val(Av(A), op(a_1, a_2, \cdots, a_k)) = op(Val(Av(A), a_1), Val(Av(A), a_2), \cdots, Val(Av(A), a_k))$
$$\text{if} \quad a_1, a_2, \cdots, a_k \in AExp(A)$$

またObTS実行中において, 属性論理式 $exp \in BExp(A)$ と属性値集合 $Av(A)$ が与えられたとき、exp 中で使われている属性の値はすべて $Av(A)$ 中に与えられており、exp 中に現れる属性式の値も評価可能なので exp の値を評価することができる。その評価した値を $Val(Av(A), exp)$ と書くことする。

D_Aを属性値のドメイン集合、R は D_A上の k 項関係演算子としたとき

定義 4.5 $Val(Av, exp)$

1. $Val(Av(A), \text{true}) = \text{true}$
2. $Val(Av(A), \text{false}) = \text{false}$
3. $Val(Av(A), R(a_1, a_2, \cdots, a_k)) = R(Val(Av(A), a_1), Val(Av(A), a_2), \cdots, Val(Av(A), a_k))$
$$\text{if} \quad a_1, a_2, \cdots, a_k \in AExp(A)$$
4. $b_1, b_2 \in BExp(A)$ のとき

$$\begin{aligned} Val(Av(A), b_1 \wedge b_2) &= Val(Av(A), b_1) \wedge Val(Av(A), b_2) \\ Val(Av(A), b_1 \vee b_2) &= Val(Av(A), b_1) \vee Val(Av(A), b_2) \\ Val(Av(A), \neg b_1) &= \neg Val(Av(A), b_1) \end{aligned}$$

4.2.2 イベントに関する定義

イベントの集合を E としたとき、イベント論理式の集合 $EExp(E)$ は次のように定義される。

定義 4.6 $EExp(E)$

1. $e \in E$ ならば $e, \neg e \in EExp(E)$
2. $exp_1, exp_2 \in EExp(E)$ ならば $exp_1 \wedge exp_2, exp_1 \vee exp_2 \in EExp(E)$

イベント集合 $E_1 \subseteq E$ に対してイベント論理式 $exp \in EExp(E)$ が真であることを $E_1 \mapsto exp$ と書く。

定義 4.7 $E_1 \mapsto exp$

1. $\forall e \in E_1,\ E_1 \mapsto e$
2. $\forall e \notin E_1,\ E_1 \mapsto \neg e$
3. $\forall exp_1, exp_2 \in EExp(E),\ E_1 \mapsto exp_1 \vee exp_2 \iff E_1 \mapsto exp_1 \vee E_1 \mapsto exp_2$
4. $\forall exp_1, exp_2 \in EExp(E),\ E_1 \mapsto exp_1 \wedge exp_2 \iff E_1 \mapsto exp_1 \wedge E_1 \mapsto exp_2$

定義 4.8 イベント e に付加されている属性集合を

$$att(e)$$

と書く。

また、イベント論理式 $exp \in EExp(E)$ が与えられたとき、exp 中のイベントの属性集合 $Ae(exp)$ は次のように定義される。

定義 4.9 $Ae(exp)$

1. $e \in E$ ならば $Ae(e) = att(e)$
2. $e \in E$ ならば $Ae(\neg e) = \emptyset$
3. $exp_1, exp_2 \in EExp(E)$ ならば $Ae(exp_1 \wedge exp_2) = Ae(exp_1) \cup Ae(exp_2)$
4. $exp_1, exp_2 \in EExp(E)$ ならば $Ae(exp_1 \vee exp_2) = Ae(exp_1) \cap Ae(exp_2)$

4.2.3 ObTS とオブジェクトの定義

定義 4.10 $ObTS$ M は $M = (O, E, root)$ のような 3 つ組で定義される。このとき

O はオブジェクトの集合
E はイベントの集合
$root \in O$ は最初に動作するオブジェクト

である。

定義 4.11 オブジェクト $Ob \in O$ は $Ob = (Attr, States, Obs, Para, Rules, p_0, E_{io})$ のように定義される。このとき

$Attr$	Ob の属性の集合
$States$	Ob の状態の集合
Obs	Ob の内部オブジェクトの集合
$Para$	並行動作する内部オブジェクトの集合類
	$Para \subset 2^{Obs}$ かつ $(x, y \in Para \wedge x \neq y \rightarrow x \cap y = \emptyset)$
$Rules$	遷移規則の集合
p_0	Ob の初期遷移で、$p_0 \in Rules$
E_{io}	Ob が入出力に使用するイベントの名前の集合 $E_{io} \subseteq E$

である。ただし複数のオブジェクトに渡って議論するときには $Attr, States, Obs, Para, Rules, E_{io}$ はそれぞれ $Attr(Ob), States(Ob), Obs(Ob), Para(Ob), Rules(Ob), E_{io}(Ob)$ のようにオブジェクト名を付加させて書くこととする。

オブジェクト $Ob = (Attr, States, Obs, Para, Rules, p_0, E_{io})$ において、$Para$ に含まれない内部オブジェクトの集合を Obs^- と書く。

定義 4.12 Obs^-

$$Obs^- = Obs - \bigcup_{a \in Para} a$$

オブジェクト $Ob = (Attr, States, Obs, Para, Rules, p_0, E_{io})$ において、状態遷移の遷移元、遷移先になるものの集合を$\widetilde{S}$と書く。$\widetilde{S}$は状態、単一の内部オブジェクトおよび並行動作する内部オブジェクトの集合からなっている。

定義 4.13 $\widetilde{S}$

$$\widetilde{S} = States \cup Obs^- \cup Para$$

状態遷移の対象 $x \in \widetilde{S}$が与えられたとき x 中のオブジェクトの集合を $OS(x)$ と書くことする。

定義 4.14 $OS(x)$

$$OS(x) \iff \begin{cases} \{\} & \text{if } x \in States \\ \{x\} & \text{if } x \in Obs^- \\ x & \text{if } x \in Para \end{cases}$$

オブジェクトの階層構造においてオブジェクト Ob の親オブジェクトは $Parent(Ob)$ と書くこととする。

定義 4.15 $Parent$

$$Parent(Ob) = \begin{cases} O_0 & \exists O_0 . Ob \in Obs(O_0) \\ \text{undefined} & \text{otherwise} \end{cases}$$

オブジェクト Ob の遷移規則において参照可能な属性の集合を $A_{ref}(Ob)$ と書くこととする。

定義 4.16 A_{ref}

$$A_{ref}(Ob) = \begin{cases} Attr(Ob) \cup A_{ref}(Parent(Ob)) & \text{if } Parent(Ob) \text{ is defined} \\ Attr(Ob) & \text{if } Parent(Ob) \text{ is undefined} \end{cases}$$

オブジェクト $Ob = (Attr, States, Obs, Para, Rules, p_0, E_{io})$ において、遷移 $p \in Rules$ は次のように定義される。

定義 4.17 状態遷移

$$p: \quad X_1 \to X_2 \ ; \ E_1[Cond]/E_2[Eval]$$

ただし、

$X_1, X_2 \in \widetilde{S}$はそれぞれ遷移元、遷移先

$E_1, E_2 \subseteq E_{io}$はそれぞれ入力と出力のイベント集合

$Cond \in BExp(A_{ref}(Ob) \cup \{att(e) | e \in E_1\})$ は遷移を実行するための属性値の条件

$Eval$ は遷移を実行する際の属性評価式のならび

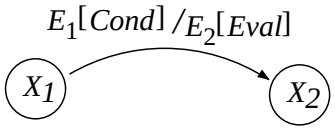

Figure 4.5: 状態遷移

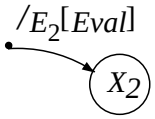

Figure 4.6: 初期遷移

である。また E_1にはイベント論理式を書くことも可能で、この場合は

$E_1 \in EExp(E_{io})$

$Cond \in BExp(A_{ref}(Ob) \cup Ae(E_1))$

となる。また、状態遷移図には図 4.5のように書く。

遷移の動作としては、X_1で入力 E_1を受け取ったとき条件 $Cond$ が成り立っていたら、X_2に遷移して E_2 を出力し、$Eval$ 中の式を計算する。

オブジェクト $Ob = (Attr, States, Obs, Para, Rules, p_0, E_{io})$ において、初期遷移 $p_0 \in Rules$ は次のように定義される。

定義 4.18 初期遷移

$$p_0 : \quad Ob \to X_2 \ ; \ /E_2[Eval]$$

ただし、

$X_2 \in \widetilde{S}$は遷移先

$E_2 \subseteq E_{io}$は出力イベント集合

$Eval$ は属性評価式のならび

である。初期遷移は各オブジェクトに 1 つだけ定義される。また、状態遷移図には図 4.6のように書く。

オブジェクト $Ob = (Attr, States, Obs, Para, Rules, p_0, E_{io})$ の状態遷移

$$p : \quad X_1 \to X_2 \ ; \ E_1[Cond]/E_2[Eval]$$

において、

定義 4.19 属性評価式は

$$a := exp$$

のように書く。ただし、

a は属性で $a \in Attr \cup \{att(e) | e \in E_2\}$

exp は属性式で $exp \in \begin{cases} AExp(A_{ref}(Ob) \cup \{att(e) | e \in E_1\}) & \text{if } \ E_1 \subseteq E_{io} \\ AExp(A_{ref}(Ob) \cup Ae(E_1)) & \text{if } \ E_1 \in EExp(E_{io}) \end{cases}$

である。

4.3 Statechart 式通信モデルに基づくセマンティクス

ObTS を Statechart 式通信モデルを用いて動作させるためのセマンティクスは以下のように与えられる。

4.3.1 オブジェクトコンフィギュレーション

Statechart 式通信モデルにおいて、オブジェクトの動作状態 $OC = (OX, OA)$ をオブジェクトコンフィギュレーションと言い、このとき OX, OA はそれぞれ

OX　現在の状態または単一の内部オブジェクトまたは並行動作する内部オブジェクトの集合で $OX \in \widetilde{S}$

OA　現在の属性値集合で $OA = Av(A \cup A_{ref}(Ob))$

である。オブジェクト x のオブジェクトコンフィギュレーションを $OC_x = (OX_x, OA_x)$ と書くこととする。

オブジェクト x の初期遷移が実行されてオブジェクトコンフィギュレーション $OC = (OX, OA)$ となりイベント集合 E_{out}が出力されることを

$$x \to (OC, E_{out})$$

と書くこととする。x の初期遷移が

$$x \to X_2 \ ; \ /E_2[a_1 := exp_1; a_2 := exp_2; \cdots; a_n := exp_n]$$

とすると、

定義 4.20 $x \to (OC, E_{out})$

$$x \to (OC, E_{out}) \iff \begin{cases} OX = X_2 \\ OA = \{(a_i, Val(Av(A_{ref}(Ob)), exp_i)) | 1 \leq i \leq n\} \\ E_{out} = E_2 \end{cases}$$

$OC = (OX, OA)$ であるようなオブジェクトにイベント集合 E_{in}が与えられて遷移したとき、新しいオブジェクトコンフィギュレーション $OC' = (OX', OA')$ となりイベント集合 E_{out}が出力されることを

$$(OC, E_{in}) \to (OC', E_{out})$$

と書くこととする。

$$X_1 \to X_2 \ ; \ E_1[Cond]/E_2[a_1 := exp_1; a_2 := exp_2; \cdots; a_n := exp_n]$$

のような遷移規則が与えられたとき

定義 4.21 $(OC, E_{in}) \to (OC', E_{out}) \iff$

$$\begin{cases} X_1 = OX \\ E_1 \subseteq E_{in} \vee E_{in} \mapsto E_1 \\ Val(OAE, Cond) = \text{true} \\ OX' = X_2 \\ OA' = OA[a_1/Val(OAE, exp_1)][a_2/Val(OAE, exp_2)] \cdots [a_n/Val(OAE, exp_n)] \\ E_{out} = E_2 \\ \text{where } OAE = OA \cup Av(Ae(E_1)) \end{cases}$$

4.3.2 システムコンフィギュレーション

Statechart 式通信モデルにおいて、ObTS の動作状態 $SC = (SX, SE)$ をシステムコンフィギュレーションと言い、SX, SE はそれぞれ

SX　現在動作中のオブジェクトの集合で $SX \subseteq \{(x, OC_x) | x \in O\}$

SE　現在有効なイベントの集合で $SE \subseteq E$

である。

ObTS の初期システムコンフィギュレーションは root オブジェクトの初期遷移を実行したあとのシステムコンフィギュレーションである。ただし、root オブジェクトの初期遷移によって新たに起動されるオブジェクトがあれば、そのオブジェクトの初期遷移も実行する。それによって更に起動されるオブジェクトがあればその初期遷移も実行し、起動されるオブジェクトがなくなるまで繰り返す。起動されたすべてのオブジェクトの集合と、それらの初期遷移によって出力されたイベントの集合が初期システムコンフィギュレーションになる。つまり、

定義 4.22 ObTS の初期システムコンフィギュレーション $SC_0 = (SX_0, SE_0)$ は

$$\begin{aligned} SE_0 &= \bigcup_{0 \le i < n} \{E | (x, OC, E) \in T_i\} \\ SX_0 &= \bigcup_{0 \le i < n} \{(x, OC) | (x, OC, E) \in T_i\} \\ \text{where} \quad T_0 &= \{(root, OC_{root}, E_{root})\} \quad \text{ただし } root \to (OC_{root}, E_{root}) \\ T_i &= \{(OX, OC', E') | (x, OC, E) \in T_{i-1} \wedge OX \in O \wedge OX \to (OC', E')\} \\ &\quad \cup \left[\bigcup_{(x, OC_x, E) \in T_{i-1} \wedge OX_x \subseteq O} \{(y, OC_y, E_y) | y \in OX_x \wedge y \to (OC_y, E_y)\} \right] \\ n &\quad \text{は } T_n = \{\} \text{ かつ } T_{n-1} \neq \{\} \text{ であるような整数} \end{aligned}$$

である。

上の定義において T_0 というのは root オブジェクトの初期遷移のみを実行した時点での、動作中のオブジェクト (root だけ) とそのオブジェクトコンフィギュレーションと遷移によって出力されたイベントとの 3 つ

組の集合である。root オブジェクトの初期遷移の遷移先が状態であれば、そのまま $T_1 = \{\}, n = 1$ となるが、遷移先が内部オブジェクトもしくは内部オブジェクトの集合であった場合にはそれらのオブジェクトの初期遷移も実行する必要がある。それらの初期遷移を実行した時点で新たに動作中になったオブジェクトについて T_0 の場合と同じように 3 つ組を集めた集合が T_1 となる。T_i の式の前半は単独で起動されるオブジェクトについてのもので、後半は並行動作する複数のオブジェクトが起動される場合のものである。新たに起動されるオブジェクトがなくなるまで T_i を求めていき、出力されたすべてのイベントの集合が SE_0、起動されたすべてのオブジェクトとそのオブジェクトコンフィギュレーションの集合が SX_0 となる。

Statechart 式通信モデルにおける ObTS の動作は SC の遷移で記述できる。$SC = (SX, SE)$ から遷移して $SC' = (SX', SE')$ に変わることを

$$SC \rightarrow SC'$$

と書くこととする。SC から SC' への遷移は

1. SX 中のオブジェクトに SE を入力として与えて、遷移可能なオブジェクトをそれぞれ 1 回ずつ遷移させる。

2. 1での遷移によって新たに起動されたオブジェクトがあれば、それらのオブジェクトの初期遷移を実行する。

3. 2での遷移によって新たに起動されたオブジェクトがあれば、それらのオブジェクトの初期遷移を実行する。新たに起動されるオブジェクトがなくなるまで、これを繰り返す。

4. 1から 3で出力されたすべてのイベントの集合を SE' とする。

5. SX 中のオブジェクトおよび新たに起動されたオブジェクトの集合から、1の遷移によって終了したオブジェクトを除いたものを SX' とする。もちろん遷移したオブジェクトについては遷移後の新しいオブジェクトコンフィギュレーションとのペアで SX' に入れる。

というような動作になる。これをまとめると次のようになる。

定義 4.23 $SC \to SC' \iff$

$$
\begin{cases}
SE' & = \bigcup_{0 \le i < n} \{E | (x, OC, E) \in T_i\} \\
SX' & = \{(x, OC) | (x, OC) \in SX \wedge (x, OC', E) \notin T_0\} \\
 & \quad \cup \left[\bigcup_{0 \le i < n} \{(x, OC) | (x, OC, E) \in T_i\} \right] \\
 & \quad - \{(y, OC_y) | (x, OC) \in SX \wedge (x, OC', E) \in T_0 \wedge \\
 & \qquad\qquad (y = OX \wedge y \neq OX' \vee y \in OX \wedge y \notin OX')\} \\
\text{where} & \quad T_0 = \{(x, OC', E) | (x, OC) \in SX \wedge (OC, SE) \to (OC', E)\} \\
 & \quad T_i = \{(OX, OC', E') | (x, OC, E) \in T_{i-1} \wedge OX \in O \wedge OX \to (OC', E')\} \\
 & \quad \cup \left[\bigcup_{(x, OC_x, E) \in T_{i-1} \wedge OX_x \subseteq O} \{(y, OC_y, E_y) | y \in OX_x \wedge y \to (OC_y, E_y)\} \right] \\
 & \quad n \text{ は } T_n = \{\} \text{ かつ } T_{n-1} \neq \{\} \text{ であるような整数}
\end{cases}
$$

上の定義においてT_0はSEによって直接遷移を起こすオブジェクトと、遷移後のオブジェクトコンフィギュレーションと、遷移で発生したイベントのタプルの集合である。T_1については先に述べたSC_0の場合と同様で、T_0の遷移によって新たに起動されるオブジェクトと、そのオブジェクトの初期遷移後のオブジェクトコンフィギュレーションと、初期遷移によって発生したイベントのタプルの集合である。以降、新たに起動されるオブジェクトがなくなるまでT_iを求めていき、出力されたすべてのイベントの集合がSE'、SEで遷移したオブジェクトと新たに起動されたオブジェクトの集合がSX'となる。

第 5 章

仕様記述言語 ObCL とシミュレーション環境

本章では ObTS モデルのための記述言語 ObCL を導入し、関数型言語による ObTS シミュレータおよび ObCL 記述支援環境について述べる。

5.1 ObCL

ObTS モデルでは実際にモデルを記述するための言語を規定しておらず、また記述の再利用などに係わる継承などの概念が導入されていない。そこで本節では ObTS モデルに基づいて仕様記述をするための言語の提案および記述の便宜のための概念の導入を行う。

5.1.1 概要

ObCL は ObTS モデルが持つ各概念に加え、記述の再利用性や可読性の向上のためにクラス、継承、マルチキャスト通信、イベントクラス等の概念を導入した言語である。

- システム
  ObCL ではクラス/フィールド/イベントクラスの集まりによってシステムを記述する。
- クラス
  ObTS モデルではオブジェクトの集合としてシステムを捉えているが、ObCL では記述の再利用等を考慮して ObTS でのオブジェクトをクラスで記述し、ObCL 記述を ObTS 化する際にクラスをインスタンス化してオブジェクトを得る。具体的には ObTS のオブジェクト階層に対応するクラス階層を ObCL で記述し、クラス記述に含まれる内部オブジェクト記述 (内部オブジェクト名とクラス名の対からなる) にしたがってインスタンス化する。

一般のオブジェクト指向言語でのクラス記述はインタフェース部と実現部の記述に分かれており、実現部は隠蔽されているが、ObTS のようにオブジェクトの動作を状態遷移図のみで記述する場合には、インタフェース部に相当する部分をあげるとすると、状態遷移の入力イベントの部分やそれに対して他のオブジェクトに対して出力するイベントの部分ということになる。

ObCL ではそもそもイベント通信でしか他のクラスに対して作用を及ぼせられないので、記述上でインタフェース部と実現部を明示的に分ける必要がない。

クラスはフィールド宣言/属性/状態/内部オブジェクト宣言/(操作)/遷移規則の各節で定義される。

- クラスの継承

  クラスの継承を導入することで似通ったクラス間で仕様記述を共有する。ここでの継承とは textual な記述の再利用のためのものであり、クラス間の共通部分を意味的には不完全なクラスとして記述しておき、その共通部分クラスの拡張として他のクラスを記述するというものである。継承元の全節を引き継いだ上でフィールド宣言/属性/状態/(操作)/遷移規則の追加ができるものとする。

- マルチキャスト通信 (フィールドの導入)

  ObTS モデルでは Statechart と同様にイベント通信はブロードキャストのみであるが、この言語では関連のあるクラス間に限定したマルチキャスト通信を行うこととし、この通信範囲をフィールドとして定義する。イベントの送受信はフィールドを通じた間接通信によって行う。

- イベントクラス

  ObTS ではイベントは属性を持つが、イベント属性に対する情報隠蔽のため、この言語では属性に対する操作も持たせ、操作を通じてのみその属性へのアクセスが可能であるものとする (予定)。さらにイベントに関する記述の再利用のためイベントクラスを導入し、イベントはイベントクラスのインスタンスとして定義する。

- 動的に要素数の変わる均質なインスタンスの集まり (予定)

  ObCL では ObTS と同様にインスタンス (オブジェクト) の動的生成はしない。

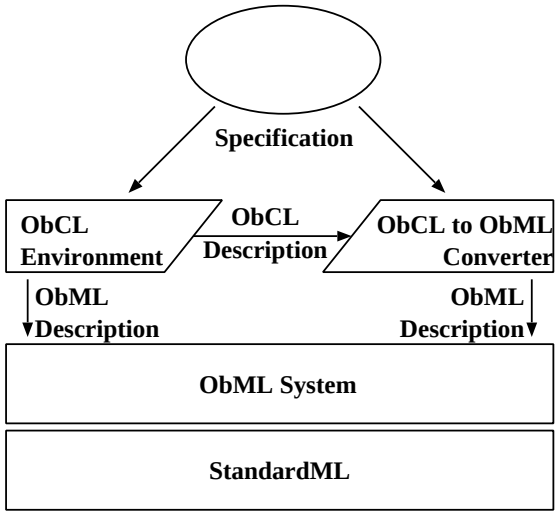

Figure 5.1: ObTS/ObCL 記述支援/シミュレーション環境

5.1.2 言語

ObCL 言語の文法の概略を BNF ライクな記法で表すと以下のようになる。

記述	::=	(イベントクラス | フィールド | クラス)*
識別子リスト	::=	識別子 [“,” 識別子リスト]
フィールド	::=	“FIELD” 識別子 “EVENT” 識別子リスト “:” イベントクラス名 (“;” 識別子リスト “:” イベントクラス名)* “END”
イベントクラス	::=	“EVENT” 識別子 [“INHERIT” 識別子] [属性] “END”
クラス	::=	“CLASS” 識別子 [“INHERIT” 識別子] (フィールド参照 | 属性 | 状態 | 内部オブジェクト | 遷移)* “END”
フィールド参照	::=	“FIELD” 識別子リスト
属性	::=	“ATTRIBUTE” 識別子リスト “:” 型 (“;” 識別子リスト “:” 型)*
状態	::=	“STATE” 識別子リスト
内部オブジェクト	::=	“INNER” 内部オブジェクト宣言 (“,” 内部オブジェクト宣言)*
内部オブジェクト宣言	::=	(識別子リスト “:” クラス名 | 識別子 “:{” 識別子リスト “:” クラス名 (“;” 識別子リスト “:” クラス名)* “}”)
遷移	::=	“TRANSITION” 遷移規則 (遷移規則)*
遷移規則	::=	識別子 “IS” “SOURCE” (“INIT”| 識別子 “INPUT” イベント式 [“WHEN” 論理式]) “DO” 属性名 “:=” 式 (“;” 属性名 “:=” 式)* “DESTINATION” 識別子 [“OUTPUT” イベント式] “END”

5.2 ObCL 記述支援/シミュレーション環境

関数型言語 StandardML 上で ObCL による仕様記述、および、それをインスタンス化して ObTS レベルの ML コードに落してその動作をシミュレーションする環境を構築した (図 5.1)。

この環境は

- ObCL 記述支援環境
- ObCL → ObML コンバータ
- ObTS シミュレーション環境 ObML

から成る。

5.2.1 ObCL → ObML コンバータ

これは ObCL による記述を後述の ObML 上で実行できるコードに変換するコンバータである。コンバータ自身は ML とは別に書かれており単独で動作する。このコンバータを用いて ObML コードを生成する場合は ObCL 記述支援環境を使用する必要はない。

5.2.2 ObCL 記述支援環境

関数型言語 StandardML 上で ObCL のクラスおよびその部分を表す型を用意し、クラスを対話的またはバッチ処理的に記述するための ML 関数群と、そこで完成されたクラスを ObCL でのクラス記述に戻したり、それをインスタンス化して ML 上でシミュレーション可能なコードを生成したりできる記述支援環境を作成した。この環境下では記述だけでなくクラスを構成していく過程を ML の関数などに残して再利用することが可能となる。この環境についての詳細はここでは述べないが、ここで使われる型や関数についてその概略を述べる。

型 この環境で用いられる主な型は表 5.1の通りである。状態遷移の条件や属性評価のための『式』は属性参照部分以外には ML の任意の式が扱えるように、環境下での型は単純なものになっている。『属性評価式』は評価される属性の名前と『式』からなり、『遷移規則』は規則名/遷移元/入力イベント名/条件式/属性評価式/遷移先/出力イベント名からなる (Assign 型は単一の属性評価式ではなく属性評価式のリストの型であることに注意する) 。『属性宣言』は属性名とその型からなる。ここでは型として ML の整数型か文字列型しか扱えないが、ML で使える任意の型を使えるように ObCL 環境を容易に拡張できる。『イベントクラス』はイベントクラス名と属性宣言リストからなり、『イベント宣言』はイベント名とイベントクラスからなる。『フィールド宣言』はフィールド名とイベント宣言リストからなる。『内部オブジェクト宣言』は、それが単独で動作する内部オブジェクトであればその名前とクラス名からなり、並行動作する内部オブジェクト集合であればそれぞれの名前とクラス名の

Table 5.1: 記述支援のための主要な型

```
(* 式 *)
datatype Exp_type
  = Attr of string | EInt of int | EString of string | Operation of string
type AExp = Exp_type list
type EExp = Exp_type list

type Assign = (string * AExp) list (* 属性評価式 *)

(* 遷移規則 *)
type Trans = Name * string * EExp * AExp * Assign * string * EExp

type Attr_decl = Name * Attribute_type (* 属性宣言 *)
type EventClass = Name * Attr_decl list (* イベントクラス *)
type Event_decl = Name * EventClass (* イベント宣言 *)
type Field = Name * Event_decl list (* フィールド宣言 *)

(* 内部オブジェクト宣言 *)
datatype Inner_type
  = IParallel of (string * Class) list | ISingle of Class
type Inner_decl = Name * Inner_type

(* クラス *)
type Class =
  ClassName * Field list * Attr_decl list * string list * Inner_decl list
  * Trans list
```

リストと、その並行動作する集合全体の名前からなる。『クラス』はクラス名/属するフィールド名リスト/属性宣言リスト/状態名リスト/内部オブジェクト宣言リスト/遷移規則リストからなる。ObCL 環境ではクラスの継承はクラスを記述していく過程の 1 つの手順として扱われ、ML 中のデータとしては扱われないのでクラス型には継承に関する情報は含まれない。

関数 この環境で用いられる主な関数の signature は表 5.2の通りである。FIELD に含まれる関数はイベントクラスおよびフィールドの記述に関するもの、CLASS はクラスの記述に関するもの、OBCLOUT と INSTANCE はそれぞれ記述された ML データを ObCL 記述で出力するものと ObML 用の ML コードで出力するものである。

クラス記述に関しては newclass と newtrans はそれぞれ名前だけを持つクラスや遷移規則のテンプレートを返し、これに append で始まる各関数を用いて記述を追加していくことでクラス記述を構成していく。クラスの継承には extendclass を用い、継承元のクラスデータとそれによって作られる新たなクラスの名前を指定する。

5.2.3 ObTS シミュレーション環境 ObML

ObTS モデルを計算機上で動かすためのシミュレータを関数型言語 Standard ML で作成した。このシミュレータでは ObTS モデル自身も ML の関数群として記述し、シミュレータ本体と合わせて実行する。動作結果は各ステップにおける状態とイベントに関するスナップショット (システムコンフィグレーション)

Table 5.2: 記述支援のための主要な関数

```
signature FIELD=
  sig
    val appendattrtoevclass : EventClass -> Attr_decl list -> EventClass
    val appendeventtofield : Field -> Event_decl list -> Field
    val makeeventclass : Name -> Attr_decl list -> EventClass
    val extendeventclass : Name -> EventClass -> EventClass
    val generic_event : EventClass

    val makefield : Name -> Event_decl list -> Field
    val neweventclass : Name -> EventClass
    val newfield : Name -> Field
  end

signature CLASS =
  sig
    val newassign : string * AExp -> Assign
    val newexp : string -> AExp

    (* 遷移規則関係 *)
    val appendassn : Trans * Assign -> Trans
    val appendcond : Trans * AExp -> Trans
    val appenddest : Trans * string -> Trans
    val appendinput : Trans * EExp -> Trans
    val appendoutput : Trans * EExp -> Trans
    val appendsrc : Trans * string -> Trans
    val createtrans : Name * string * EExp * AExp * Assign * string * EExp
                      -> Trans
    val inittrans : Name * Assign * string * EExp -> Trans
    val newtrans : Name -> Trans

    (* クラス関係 *)
    val appendattribute : Class * Name list * Attribute_type -> Class
    val appendfield : Class * Field list -> Class
    val appendparallel : Class * Name * (Name * Class) list -> Class
    val appendsingle : Class * Name list * Class -> Class
    val appendstate : Class * string list -> Class
    val appendtrans : Class * Trans list -> Class
    val extendclass : ClassName * Class -> Class
    val newclass : ClassName -> Class
  end

signature OBCLOUT=
  sig
    val obclclass : Class -> string
    val obcltrans : Trans -> string
    val obclevclass : EventClass -> string
    val obclfield : Field -> string
    val obclall : Class -> string
  end

signature INSTANCE=
  sig
    val instanciate : string * Name * Class -> string
    val instanciate_recursive : string * Name * Class -> string
  end
```

Table 5.3: シミュレーションのための主要な型

```
(* 属性とその値 *)
datatype Attribute_type = Int of int | Null_of_Attr | String of string
type  Attribute = Name * Attribute_type

(* イベント *)
type  Event = Name * Attribute list

(* 環境 = (オブジェクト名 * 属性値集合) 集合 *)
type  Env = (Name * Attribute list) list

(* 遷移対象: 状態/内部オブジェクト/並行内部オブジェクト集合 *)
datatype X
  = Initial
  | Name
  | Null
  | Parallel of Object list
  | Single of Object
  | State of string

(* オブジェクト *)
type  Object =
  X * Attribute list * Event list * Env -> X * Attribute list * Event list

(* オブジェクトコンフィグレーション *)
type  ObjectConf = Object * X * Attribute list

(* システムコンフィグレーション *)
type  SysConf = ObjectConf list * Event list
```

のリストとして返され、必要に応じて読みやすい形に直してテキスト表示される。この動作結果のリストを ML 関数で処理することで記述対象の動作に関する性質のテストなどに利用することができる。

型 シミュレーションに関する主な型には表 5.3のようなものがある。オブジェクトは関数として定義されていることに注意する。前述の記述支援環境を使ってクラス記述したものを単にシミュレーションし、プリティプリンタ関数の出力でその動作を確認するだけであれば、必ずしもこれらの型を知っている必要はないが、SysConf 型のリストで返される実行結果を使って任意の ML 関数を用いた動的テストなどに利用できる。

関数 シミュレーションに関する主な関数は表 5.4の通りである。基本的なシミュレーションは関数 system に最初から起動すべきオブジェクトのリストと、実行したいステップ数を与えることで行われる。シミュレーション中に外部からのイベントを与えたい場合は関数 system' を用いる。また、関数 systemTrans は system' に対して起動すべきオブジェクトリストの代わりに実行し始めるシステムコンフィグレーションを指定する。systemTrans を繰り返し用いることで外部から与えるイベントをその都度指定しながら対話的に実行を進めることができる。SIMULATOR に含まれる他の関数は、関数として書かれている個々のオブジェクトから使用される補助関数である。関数 ppSystem は実行結果として出されるシステムコンフィグレーションのリストを見やすい形で表示するためのものである。

Table 5.4: シミュレーションのための主要な関数

```
signature SIMULATOR=
  sig
    val changeAttr : Name -> Attribute_type -> Attribute list -> Attribute list
    val eventAttr : Name -> Name -> Event list -> Attribute_type
    val makeEvents : Name list -> Event list
    val memberEv : Name -> Event list -> bool
    val objectAttr : Name -> Name -> Event list -> Attribute_type
    val objectName : Object -> string
    val pickupAttr : Name -> Attribute list -> Attribute_type
    val searchEv : (Event -> bool) -> Event list -> Event
    val searchEvlist : (Event -> bool) -> Event list -> Event list
    val systemTrans : SysConf * Event list list * int
                      -> SysConf list
    val system : Object list -> int -> SysConf list
    val system' : Object list
                  -> Event list list
                     -> int -> SysConf list
  end

signature PRETTY=
    sig
        val ppSystem : SysConf list -> unit
    end
```

第 6 章

記述例

ObTS および ObCL の記述例として単純なエアコン装置の例を示す。

6.1 仕様

エアコン装置の仕様は以下のようなものとする (ただし d は定数)。

- 暖房モードと冷房モードがあり、スイッチによって電源オフ／暖房／冷房が切り替わる。
- 温度アップおよび温度ダウンのスイッチにより目標温度が設定可能である。
- 暖房モードでは室温が目標温度より低いときに実際に暖房がオンになり、目標温度 $+ d$ を越えたら暖房をオフにする。
- 冷房モードでは室温が目標温度より高いときに実際に冷房がオンになり、目標温度 $- d$ を下回ったら冷房をオフにする。
- 電源がオン (つまり暖房モードか冷房モード) であることを示すインジケータがついている。

6.2 ObTS による状態遷移図記述

このエアコンを ObTS の状態遷移図で記述すると図 6.1のようになる。また、使われているイベントは表 6.1のような意味である。

エアコンはインジケータ/システムパネル/本体の 3 つの並行動作するオブジェクトからなり、本体は内部オブジェクトとして暖房/冷房の 2 つのオブジェクトを持つものとした。各オブジェクトはそれぞれのボタン押下イベントに対応して状態遷移するほか、情報隠蔽により目標温度を記憶する属性を共有できないので、システムパネル/冷房/暖房は各々目標温度を記憶する属性を持ち、目標温度のアップ/ダウンのボタ

Table 6.1: イベント表

名前	from	to	意味
clock	外部	暖房, 冷房	1 秒おきにセンサから室温が送られてくる
off	外部	本体, インジケータ	電源オフ
heat	外部	本体, インジケータ	暖房モード
cool	外部	本体, インジケータ	冷房モード
tup	外部	システムパネル	目標温度アップ
tdown	外部	システムパネル	目標温度ダウン
tset	システムパネル	暖房, 冷房	目標温度更新

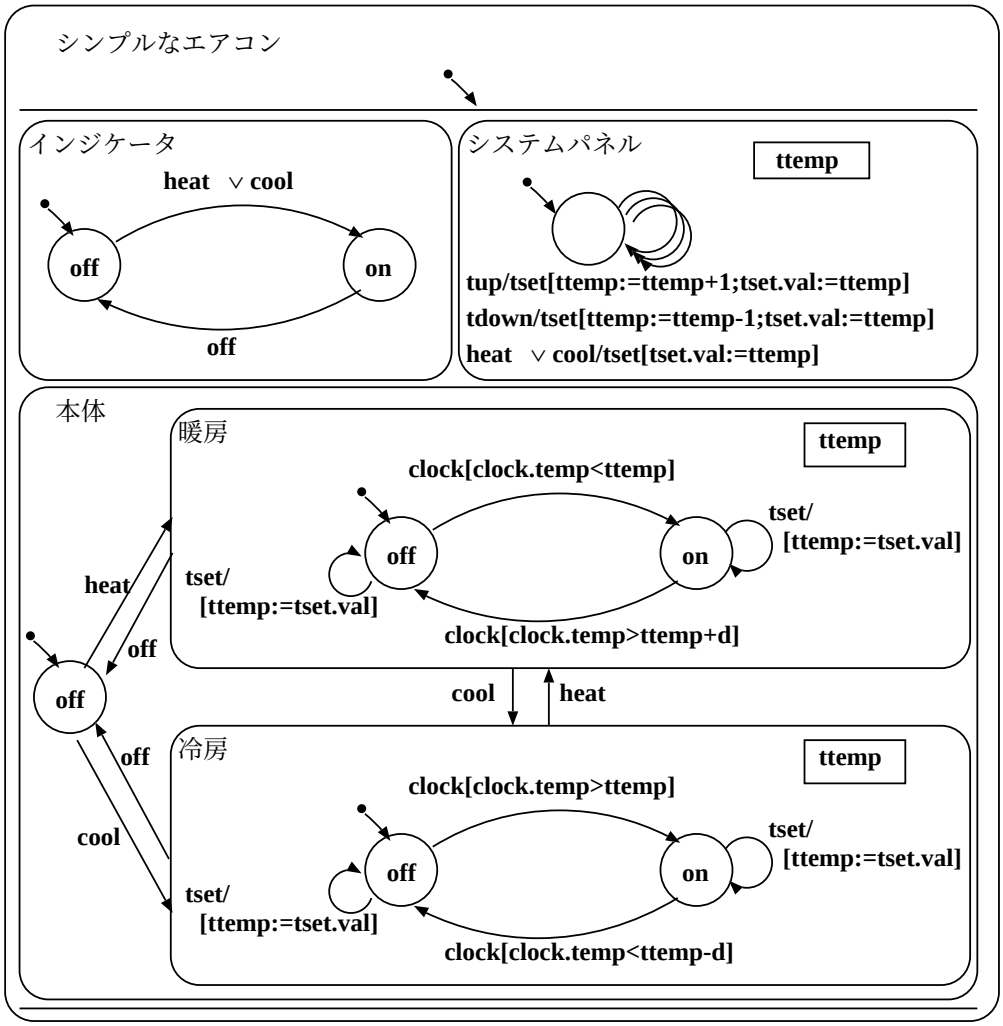

Figure 6.1: エアコンの ObTS 記述

ンが押されるたびにシステムパネルから冷房/暖房オブジェクトへ向けて目標温度変更通知のためのイベントが送られるものとした。

なお、システムパネルは単一の状態で 3 種類の遷移規則が定義されているが、図 6.1では 3 つの状態遷移を簡略化して書いてある。また、システムパネルの 3 つの状態遷移および暖房/冷房のオブジェクト中の状態遷移は非決定性を生じる可能性を持っている。本来は遷移のトリガにイベント論理式を用いることで非決定性を生じないように記述すべき箇所であるが、記述がわかりにくくならないためにここではそのまま表記した。なお、前章で紹介した ObML では実装上同一オブジェクトが複数の遷移を実行できる状況になっても、記述上先に現れる遷移が必ず優先されて実行される。

6.3 ObCL による記述

ObCL での記述は表 6.2〜6.5のようになる。

組込みのイベントクラス GENERIC_EVENT を継承して、整数型属性を 1 つ持つイベントクラス ONEARG_EVENT を作っておく。フィールドはボタン押下イベントを伝える BUTTON_FIELD、センサからの温度通知のための CLOCK_FIELD、および、エアコン内部の通信のための TSET_FIELD を定義する。ON/OFF の 2 状態と OFF 状態への初期遷移を持つ抽象クラス ONOFF_CLASS と、ONOFF_CLASS を継承して暖房と冷房の共通する遷移/属性を追加した抽象クラス ONOFF2_CLASS を作っておき、システムパネルは ONOFF_CLASS を継承して、暖房と冷房は ONOFF2_CLASS を継承して作る。

6.4 ObCL 環境での記述スクリプト

ObCL 環境下では表 6.6〜6.7のようにしてエアコンの記述を環境内のイベントクラス/フィールド/クラス型変数に用意することができる。これを関数 obclout.obclclass 等に通せば前節で示したような ObCL 記述が得られ、関数 instance.instanciate に通せば ObML で実行可能な ML コードが生成される。

ここで注目すべきなのはクラスの継承だけでなく、似たような遷移規則の記述のために ML で生成関数を用意して簡便に記述できるほか、記述そのものの再利用だけでなく記述過程を ML 関数として再利用できるということである。

Table 6.2: エアコンの ObCL 記述 (1)

```
event --- 属性を 1 つ持つイベントクラス
  ONEARG_EVENT
inherit
  GENERIC_EVENT
attribute
  val:Int
end

field --- ボタン押下イベントのフィールド
  BUTTON_FIELD
event
  heat,cool,off,tup,tdown:GENERIC_EVENT
end

field --- エアコン内部の通信のためのフィールド
  TSET_FIELD
event
  tset:ONEARG_EVENT
end

field --- 温度センサから毎秒室温が伝えられるフィールド
  CLOCK_FIELD
event
  clock:ONEARG_EVENT
end

class --- OFF/ON の 2 状態と OFF への初期遷移のある抽象 (不完全) クラス
  ONOFF_CLASS
field
  BUTTON_FIELD
state
  off,on
transition
  start is
    source      init
    destination off
    end
end

class --- インジケータクラス
  INDICATOR_CLASS
inherit
  ONOFF_CLASS
transition
  t1 is
    source      off
    input       BUTTON_FIELD.heat or BUTTON_FIELD.cool
    destination on
    end
  t2 is
    source      on
    input       BUTTON_FIELD.off
    destination off
    end
end
```

Table 6.3: エアコンの ObCL 記述 (2)

```
class --- ONOFF_CLASS に対して目標温度受信処理を追加した抽象 (不完全) クラス
  ONOFF2_CLASS
inherit
  ONOFF_CLASS
field
  TSET_FIELD, CLOCK_FIELD
attribute
  ttemp:Int
transition
  toff is
    source      off
    input       TSET_FIELD.tset
    do          ttemp := TSET_FIELD.tset.val
    destination off
    end
  ton is
    source      on
    input       TSET_FIELD.tset
    do          ttemp := TSET_FIELD.tset.val
    destination on
    end
end

class --- 暖房クラス
  HEATER_CLASS
inherit
  ONOFF2_CLASS
transition
  t1 is
    source      off
    input       CLOCK_FIELD.clock
    when        CLOCK_FIELD.clock.val < ttemp
    destination on
    end
  t2 is
    source      on
    input       CLOCK_FIELD.clock
    when        CLOCK_FIELD.clock.val > ttemp + 1
    destination off
    end
end

class --- 冷房クラス
  COOLER_CLASS
inherit
  ONOFF2_CLASS
transition
  t1 is
    source      off
    input       CLOCK_FIELD.clock
    when        CLOCK_FIELD.clock.val > ttemp
    destination on
    end
  t2 is
    source      on
    input       CLOCK_FIELD.clock
    when        CLOCK_FIELD.clock.val < ttemp - 1
    destination off
    end
end
```

Table 6.4: エアコンの ObCL 記述 (3)

```
class --- システムパネルクラス
  PANEL_CLASS
field
  BUTTON_FIELD, TSET_FIELD
state
  s1
attribute
  ttemp : Int
transition
  start is
    source      init
    do          ttemp := 20
    destination s1
    end
  t1 is
    source      s1
    input       BUTTON_FIELD.tup
    do          ttemp := ttemp + 1;
                TSET_FIELD.tset.val := ttemp
    destination s1
    output      TSET_FIELD.tset
    end
  t2 is
    source      s1
    input       BUTTON_FIELD.tdown
    do          ttemp := ttemp - 1;
                TSET_FIELD.tset.val := ttemp
    destination s1
    output      TSET_FIELD.tset
    end
  t3 is
    source      s1
    input       BUTTON_FIELD.heat or BUTTON_FIELD.cool
    do          TSET_FIELD.tset.val := ttemp
    destination s1
    output      TSET_FIELD.tset
    end
end
```

Table 6.5: エアコンの ObCL 記述 (4)

```
class --- 本体クラス
  MAIN_CLASS
field
  BUTTON_FIELD
state
  off
inner
  heat:HEATER_CLASS;
  cool:COOLER_CLASS
transition
  start is
    source      init
    destination off
    end
  theat1 is
    source      off
    input       BUTTON_FIELD.heat
    destination heat
    end
  theat2 is
    source      cool
    input       BUTTON_FIELD.heat
    destination heat
    end
  tcool1 is
    source      off
    input       BUTTON_FIELD.cool
    destination cool
    end
  tcool2 is
    source      heat
    input       BUTTON_FIELD.cool
    destination cool
    end
  toff1 is
    source      cool
    input       BUTTON_FIELD.off
    destination off
    end
  toff2 is
    source      heat
    input       BUTTON_FIELD.off
    destination off
    end
end
--- エアコン全体
system -- class
  AIRCON
object -- inner
  aircon:
  {
    main:MAIN_CLASS;
    indicator:INDICATOR_CLASS;
    panel:PANEL_CLASS
  }
transition
  start is
    source      init
    destination aircon
    end
end
```

Table 6.6: ObCL 環境でのエアコン記述スクリプト (1)

```
open class;
open lex;

val tsetevent=newexp "TSET_FIELD.tset";
val clockevent=newexp "CLOCK_FIELD.clock";
val offevent=newexp "BUTTON_FIELD.off";
val heatevent=newexp "BUTTON_FIELD.heat";
val coolevent=newexp "BUTTON_FIELD.cool";
val heatcoolevent=heatevent @ newexp "`orelse`" @ coolevent;

(* イベントクラスとフィールドの定義 *)
local
    open field;
    val e0 = extendeventclass "ONEARG_EVENT" generic_event;
    val f0 = newfield "BUTTON_FIELD";
    val e1 = map (fn e => (e,generic_event))
        ["heat","cool","off","tup","tdown"];
in
    val onearg_event = appendattrtoevclass e0 [("val",Int 0)];
    val button_field = appendeventtofield f0 e1;
    val tset_field = makefield "TSET_FIELD" [("tset",onearg_event)];
    val clock_field = makefield "CLOCK_FIELD" [("clock",onearg_event)];
end;

(* システムパネルクラス *)
local
    val as0 = newassign("ttemp",newexp "20");
    val start = inittrans("start",as0,"s1",[]);

    val as1 = newassign("TSET_FIELD.tset.val",newexp "ttemp");
    val as2 = newassign("ttemp",newexp "ttemp - 1");
    val as3 = newassign("ttemp",newexp "ttemp + 1");
    fun paneltrans (nm,ev,assn) =
        createtrans(nm,"s1",newexp ev,[],assn @ as1,"s1",tsetevent);
    val t1 = paneltrans("t1","BUTTON_FIELD.tup",as3);
    val t2 = paneltrans("t2","BUTTON_FIELD.tdown",as2);
    val t3 = paneltrans("t3","BUTTON_FIELD.heat or BUTTON_FIELD.cool",[]);

    val c0 = ("PANEL_CLASS",[button_field,tset_field],[],["s1"],[],[]);
    val c1 = appendattribute(c0,["ttemp"],Int 0);
in
    val panel_class = appendtrans(c1,[start,t1,t2,t3])
end;

(* ON/OFF の 2 状態と OFF への初期遷移のある抽象 (不完全) クラス *)
val onoff_class:Class = ("ONOFF_CLASS",[button_field],[],["off","on"],[],[]);

(* インジケータクラス *)
local
    val start = inittrans("start",[],"off",[]);
    val t1 = createtrans("t1","off",heatevent,[],[],"on",[]);
    val t2 = createtrans("t2","off",coolevent,[],[],"on",[]);
    val t3 = createtrans("t3","on",offevent,[],[],"off",[]);
    val c0 = extendclass("INDICATOR_CLASS",onoff_class);
in
    val indicator_class = appendtrans(c0,[start,t1,t2,t3])
end;
```

Table 6.7: ObCL 環境でのエアコン記述スクリプト (2)

```
(* ONOFF_CLASS に目標温度受信を追加した抽象 (不完全) クラス *)
local
    val as0 = newassign("ttemp",newexp "20");
    val start = inittrans("start",as0,"off",[]);
    val as1 = newassign("ttemp",newexp "TSET_FIELD.tset.val");
    val toff = createtrans("toff","off",tsetevent,[],as1,"off",[]);
    val ton = createtrans("ton","on",tsetevent,[],as1,"on",[]);

    val field = [tset_field,clock_field];
    val c0 = extendclass("ONOFF2_CLASS",onoff_class);
    val c1 = appendattribute(appendfield(c0,field),["ttemp"],Int 0);
in
    val onoff2_class = appendtrans(c1,[start,toff,ton])
end;

(* 暖房/冷房クラス *)
local
    fun heatcool (nm,cond1,cond2) =
        let
            val t1 = createtrans("t1","off",clockevent,cond1,[],"on",[]);
            val t2 = createtrans("t2","on",clockevent,cond2,[],"off",[]);
            val c0 = extendclass(nm,onoff2_class);
        in
            appendtrans(c0,[t1,t2])
        end;
    val cond1 = newexp "CLOCK_FIELD.clock.val < ttemp";
    val cond2 = newexp "CLOCK_FIELD.clock.val > ttemp + 1";
    val cond3 = newexp "CLOCK_FIELD.clock.val > ttemp";
    val cond4 = newexp "CLOCK_FIELD.clock.val < ttemp - 1";
in
    val heater_class = heatcool("HEATER_CLASS",cond1,cond2)
    val cooler_class = heatcool("COOLER_CLASS",cond3,cond4)
end;

(* 本体クラス *)
local
    val start = inittrans("start",[],"off",[]);
    fun threestate (a,b) =
        createtrans(concat["t",a,b],a,
                    newexp (concat["BUTTON_FIELD.",b]),[],[],b,[]);
    val theat1 = threestate("off","heat");
    val theat2 = threestate("cool","heat");
    val tcool1 = threestate("off","cool");
    val tcool2 = threestate("heat","cool");
    val toff1 = threestate("cool","off");
    val toff2 = threestate("heat","off");

    val c0:Class = ("MAIN_CLASS",[button_field],[],["off"],[],[]);
    val c1 = appendsingle(c0,["heat"],heater_class);
    val c2 = appendsingle(c1,["cool"],cooler_class);
in
    val main_class = appendtrans(c2,[start,theat1,theat2,tcool1,tcool2,toff1,toff2])
end;

(* エアコン全体 *)
local
    val start = inittrans("start",[],"aircon",[]);
    val c1 = appendparallel(newclass("AIRCON"),"aircon",
    [("main",main_class),("indicator",indicator_class),("panel",panel_class)]);
in
    val aircon = appendtrans(c1,[start])
end;
```

第 7 章

比較

Statechart に基づいたオブジェクト指向仕様記述モデルには Objectchart[4]、ObjChart[5] 、O-Chart[6] 等がある。本章ではこれらの記述モデルを ObCL や支援環境も含めた ObTS と比較する。

7.1 Objectchart

データ計算について、Objectchart では状態遷移の pre-/post-condition によって表現しているのに対し、ObTS では属性評価式で表現する。

イベント通信については、ObCL を含まない ObTS モデルのみの場合はブロードキャスト通信のみであり、Objectchart では宛先としてオブジェクト名を明記する宛先付き直接通信であるのに対して、ObCL では宛先としてフィールド名を指定する宛先付き間接マルチキャスト通信である。フィールドを用いた通信の場合、全クラスを単一フィールドに所属させ、そのフィールドに対してのみ通信を行えばブロードキャスト通信になり、また、各クラスが個々に受信するためのフィールドを持ち、送信したい相手のフィールドに所属してそこに送信すれば、Objectchart のような宛先付き直接通信と同様になる。

Objectchart ではオブジェクト間の通信をイベント送信者、イベント、イベント受信者の 3 つ組からなるサービスリクエストで考え、サービスリクエストの可能なすべてのシーケンスの集合でシステム全体のふるまいを表現する。

Objectchart では

- 新しいサービスに対応するための状態や遷移の追加
- 遷移の強化、つまり、発火条件の弱化または事後条件の強化
- invariant relationship の強化

を継承としている。ObTS には継承はないが、ObCL の記述の再利用は Objectchart の継承の 1 つ目の場合と共通する。

Objectchart では集約関係やオブジェクトの階層構造には述べていない。ObTS ではオブジェクトの階層構造に着目しており、内部オブジェクトへの動作委譲や属性のスコープ規則について言及している。

7.2 ObjChart と O-Chart

ObjChart では内部オブジェクトもすべて最初から動作しており、集約関係は親オブジェクトが内部オブジェクトに対して特定の性質を満たすかどうかをイベント通信を用いずに直接 query する権限を持つという関係になっている。

また、O-Chart での集約関係は親と子の間の関連は明示的に記述しなくとも親オブジェクトから内部オブジェクトへの通信が可能であり、親オブジェクトが受け取ったイベントに対する動作を内部オブジェクトに委譲でき、かつ、子クラスのオブジェクトの動的生成を親クラスへの操作として記述するというものである。

これらに対して ObTS では動作委譲の対象として内部オブジェクトを捉えており、動作委譲が発生したときに内部オブジェクトが起動され、委譲が終了すれば内部オブジェクトも終了する。内部オブジェクトが動作している間は常に親オブジェクトの属性は参照可能であるが、親オブジェクトとのイベントによる通信は内部オブジェクトへの動作委譲の終了の際のみ可能となる。

ObjChart ではオブジェクト間の計算やふるまいの依存関係を表現するために relation オブジェクトを導入しており、前者の表現には属性間の関数的な invariant を用い、後者の表現には状態遷移図を用いる。各オブジェクトおよびふるまい依存関係を表す relation オブジェクトの本質的な動作は決定的な状態遷移図で記述されるが、入力イベントの受信順序や出力イベントの送信順序は非決定的に決められる。また、identical なオブジェクトの順序付き集合を表現するために sequence オブジェクトを導入している。

O-Chart ではオブジェクトの動作はそれ自身も含めた特定のオブジェクトへの操作の呼び出しとして表され、イベント生成もそれに依る。

7.3 Statechart Variants

また、[2] では多くの Statechart の変形が紹介されており、状態の階層構造に基づく遷移の優先度やイベントの時間的な有効範囲などについて比較されているが、ここで紹介されている変形は基本的にオブジェクト指向とは無関係であり、その点について ObTS とは異なっている。

第 8 章

イベント依存グラフ

本章では Statechart 式通信モデルの特徴付けおよび異なる通信モデルの比較の土台としてイベント依存グラフを導入する。

8.1 イベント依存グラフの意味

オブジェクト指向方法論においてシステム全体はオブジェクトの集まりによって表され、そのふるまいは個々のオブジェクトの内部動作とオブジェクト間の通信によって表される。各オブジェクト内の動作は状態遷移図を用いて形式的に記述されているものとし、オブジェクト間の通信の仕様をイベント依存グラフを用いて記述する。

イベント依存グラフでは個々のオブジェクトの内部構造や内部状態は考えないが、オブジェクト間の通信の観点からオブジェクトの意味ある動作は受信イベントに対するリアクションとしてのイベント送信によって表されるものとする。このようなイベント送信を発火と呼ぶこととし、発火間の依存関係が外部から観測可能で、それらをグラフに記述したものをイベント依存グラフとする。 発火間の本質的でない順序/時間関係はグラフに記述しない。

状態遷移図などによる動作記述を伴うオブジェクトの集合を O とし、オブジェクト間のイベント配送のシステムを E_{sys} とする。 E_{sys} には O 中のオブジェクト間の通信モデル M と、O を M で (決定的に) 動かすためのパラメータ (オブジェクト間の優先度やオブジェクト間のイベント伝達時間など) が含まれる。E_{sys} によって O のすべての動きが決定的に定まるものとする。 イベントシステム E_{sys} の元でのオブジェクト集合 O の動作を観測開始状態 S から有限の観測時間 T の間だけ観測し、各オブジェクトの発火および発火間の依存関係をグラフに表したものをイベント依存グラフと呼ぶ。

定義 8.1 オブジェクト集合 O、イベントシステム E_{sys}、観測開始状態 S、観測時間 T から得られるイベント依存グラフを

$$Behavior(O, E_{sys}, S, T)$$

と表すこととする。

8.2 発火とその依存関係

着目するオブジェクト集合を $O = \{o_1, o_2, \cdots, o_m\}$ とし、それらをある有限時間だけ動作させたときに観測されたすべてのイベントの集合を $E = \{e_1, e_2, \cdots, e_n\}$ とする。

定義 8.2 イベント インスタンス

同じイベントの異なる出現を区別するため、イベント名と整数の 2 つ組でイベント インスタンスを表現することとする。 つまり、イベント インスタンスは $e \in E,\ n \in N$ のとき

$$\overline{e} = (e, n)$$

のように表される。 また、このときイベント インスタンスのイベント名の部分を

$$Name(\overline{e}) = e$$

のように書くとする。 また、イベント インスタンスの集合を

$$EI = \{\overline{e_1}, \overline{e_2}, \cdots, \overline{e_l}\} \subseteq \{(e, n) | e \in E, n \in N\}$$

とする。

定義 8.3 発火

オブジェクト $o \in O$ が、イベント インスタンス $\overline{e_1}, \overline{e_2}, \cdots, \overline{e_k} \in EI$ を出力することを

$$f = (o, \{\overline{e_1}, \overline{e_2}, \cdots, \overline{e_k}\})$$

のように表し、これを「発火」と呼ぶ。単一のイベントのみを出力する通信モデルではイベント インスタンスは singleton set で表す。 また、このとき

$$Obj(f) = o$$
$$Ei(f) = \{\overline{e_i} | 1 \leq i \leq k\}$$
$$Ev(f) = \{Name(\overline{e_i}) | 1 \leq i \leq k\}.$$

と書くこととする。

定義 8.4 発火の依存関係

$\overline{E_1}, \overline{E_2}$がイベント インスタンスの集合で、発火 $f_1 = (o_1, \overline{E_1})$ から発火 $f_2 = (o_2, \overline{E_2})$ への依存関係が観測されるとき、これを $f_1 \rightsquigarrow f_2$と書く。 また、$\rightsquigarrow$ の推移的閉包を$\rightsquigarrow^+$で表す。発火間の観測される依存関係は以下のように定義される。 $Obj(f_1) = o_1$である発火 f_1は、o_1が先に発火をして以降に o が受け取ったイ

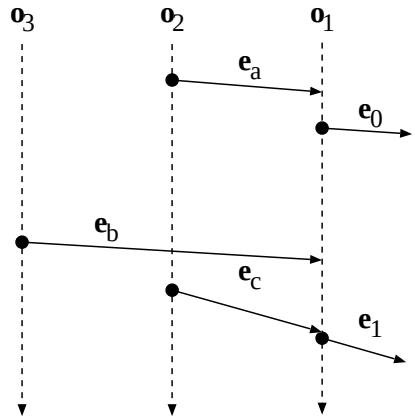

Figure 8.1a: イベントトレース図

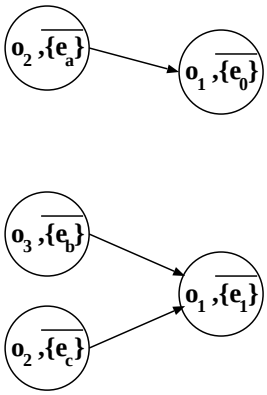

Figure 8.1b: イベント依存グラフ

Figure 8.1: 例 A

ベントインスタンスの集合$\overline{E}$に対するリアクションであると考え、 $\overline{E}$に含まれるイベントインスタンスを出力したすべての発火からの依存関係があるとする。つまり、

$$\forall f.\ \ Ei(f) \cap \overline{E} \neq \emptyset \ \Rightarrow\ f \rightsquigarrow f_1.$$

である。

例えば図 8.1aのようなイベントトレース図を考える。 ここでは o_1, o_2, o_3の 3 つのオブジェクトが動作していて、o_1は o_2の出力したイベント e_aを受け取った後、イベント e_0を出力し、o_3の出力イベント e_bと o_2の出力イベント e_cを受け取った後、イベント e_1を出力する。 これをイベント依存グラフで表すと図 8.1bのような依存関係になる。 o_1は発火 $(o_1, \overline{\{e_0\}})$ と $(o_1, \overline{\{e_1\}})$ の間にイベント e_b, e_cを受け取っているので、発火 $(o_3, \overline{\{e_b\}})$ と $(o_2, \overline{\{e_c\}})$ の両方からの依存関係がある。 また $(o_1, \overline{\{e_0\}})$ は $(o_2, \overline{\{e_a\}})$ からの依存関係がある。

8.3 イベント依存グラフ

定義 8.5 イベント依存グラフ

オブジェクト集合 O に対するイベント依存グラフ $\mathcal{G}$ は、発火の集合 $F \subseteq \{(o, \overline{E}) | o \in O,\ \overline{E} \subseteq EI\}$ と発火間の依存関係$\rightsquigarrow$ によって定義される。

$$\mathcal{G} = (F, \rightsquigarrow)$$

ただし、イベント依存グラフは以下のような性質を満たす。

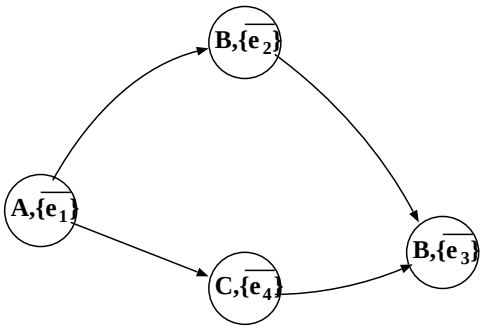

Figure 8.2a: イベント依存グラフ

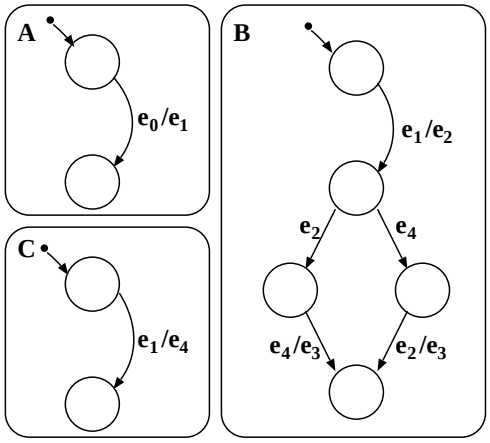

Figure 8.2b: 状態遷移図

Figure 8.2: 例 B

性質 8.1 イベント依存グラフ $\mathcal{G} = (F, \rightsquigarrow)$ 中の任意の発火は自分自身に依存関係を持たない。

$$\forall f. f \not\rightsquigarrow^+ f$$

性質 8.2 イベント依存グラフ $\mathcal{G} = (F, \rightsquigarrow)$ 中の任意の 2 つの発火は同じイベントインスタンスを持つことはない。

$$\forall f_1, f_2 \in F.\ \ f_1 \neq f_2 \Rightarrow Ei(f_1) \cap Ei(f_2) = \emptyset$$

また、イベント依存グラフ $\mathcal{G} = (F, \rightsquigarrow)$ 中の任意の発火 f に対して、f からの直接の依存関係のある発火集合を $\mathrm{Post}(f)$ と書き、その定義は以下のようになる。

定義 8.6

$$\mathrm{Post}(f) = \{f' | f \rightsquigarrow f', f' \in F\}$$

同様に、イベント依存グラフ $\mathcal{G} = (F, \rightsquigarrow)$ 中の任意の発火 f に対して、f への直接の依存関係のある発火集合を $\mathrm{Pre}(f)$ と書き、その定義は以下のようになる。

定義 8.7

$$\mathrm{Pre}(f) = \{f' | f' \rightsquigarrow f, f' \in F\}$$

例えば図 8.2bの状態遷移図 (状態名は省略) で示されるような 3 つのオブジェクトを動作させたとき得られるイベント依存グラフは図 8.2aのようになる。(このようなオブジェクト集合の場合はイベントシステムに依らず同一のイベント依存グラフが得られる)。

第 9 章

制限されたイベント依存グラフと Statechart 式通信モデル

前章で導入したイベント依存グラフを用いて Statechart 式通信モデルを特徴付ける。

9.1 グループ/分割の定義

イベント依存グラフにおいて同時に起こってもよい発火の集合をグループと定義する。つまり、グループ内のすべての発火は (直接的にも/間接的にも) 互いに依存関係がなく、かつ、同じオブジェクトに関する発火は 2 つ以上含まれていない。 発火の集合 $G \subseteq F$ がグループであることを $IsGroup(G)$ と書くとすると、グループは

定義 9.1 グループ

$$IsGroup(G) \iff \forall f_1, f_2 \in G,\ f_1 \not\rightsquigarrow^{+} f_2 \wedge Obj(f_1) \neq Obj(f_2).$$

のように定義される。また、グループ間の依存関係を次のように定義する。

定義 9.2 グループの依存関係

あるイベント依存グラフに関する 2 つのグループ G_1, G_2について

$$G_1 \rightsquigarrow G_2 \iff (\exists f_1 \in G_1, f_2 \in G_2.\ f_1 \rightsquigarrow f_2) \wedge (\forall f_1 \in G_1, f_2 \in G_2.\ f_2 \not\rightsquigarrow^{+} f_1).$$

であるとき、G_2は G_1に依存するという。これを

$$G_1 \rightsquigarrow G_2$$

と書く。

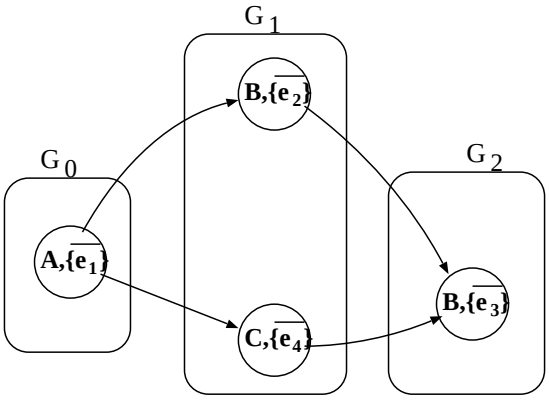

Figure 9.1: イベント依存グラフの分割

また、$\rightsquigarrow$ の推移的閉包を$\rightsquigarrow^+$で表す。

このグループの概念を用いてイベント依存グラフの分割を考える。

定義 9.3 分割

イベント依存グラフ $\mathcal{G} = (F, \rightsquigarrow)$ に対してシーケンス $\mathbf{D} = \langle G_0, G_1, \cdots, G_k \rangle$ が存在し、

$$\forall i,\ 0 \leq i \leq k.\quad IsGroup(G_i)$$

かつ

$$\forall i, j,\ 0 \leq i, j \leq k.\quad i \neq j \rightarrow G_i \cap G_j = \emptyset$$

かつ

$$F = \bigcup G_i$$

かつ

$$\forall i, j,\ 0 \leq i, j \leq k.\quad i < j \Longrightarrow\ G_i \rightsquigarrow^+ G_j$$

が成り立つとき、$\mathbf{D}$ を $\mathcal{G}$ の分割と言い、$IsPartition(\mathbf{D})$ と書く。

9.2 イベント依存グラフと Statechart 式通信モデルの対応付け

ここではどのような (条件を満たす) イベント依存グラフが Statechart 式通信モデルで実現可能であるか、また逆に Statechart 式通信モデルで実行したときに得られるイベント依存グラフはどういう条件を満たすのかを述べる。 ただし、ここではすべての状態遷移が出力イベントを持つようなオブジェクトの集合に関して述べる。 一般の場合は、イベント依存グラフについてではなくイベント依存グラフにイベントインスタンス集合が空集合である発火を追加したグラフに変換したものについて、これから述べる定理が成り立つ。

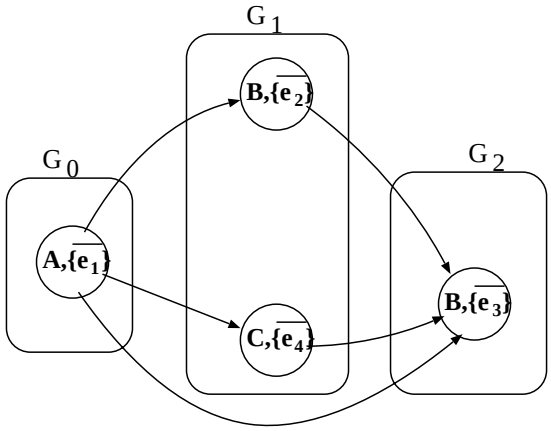

Figure 9.2: Statechart 式通信モデルで実現不可能な分割

イベント依存グラフの分割では同一のグループに含まれるすべての発火は互いに依存関係がなく、かつ、同じオブジェクトに関する発火が 2 つ以上含まれることはないので、Statechart 式通信モデルの同一ステップに起こる発火の集合との対応付けが可能である。

イベント依存グラフ $\mathcal{G}$ が次の条件$\Pi(\mathbf{D})$ を満たすような分割 $\mathbf{D}$ を持つならば $\mathcal{G}$ 中のすべてのオブジェクトについて状態遷移図を構成可能で、かつ、それらのオブジェクトを Statechart 式通信モデルで動作させたとき得られるイベント依存グラフは $\mathcal{G}$ に一致する。

定義 9.4 $\Pi(\mathbf{D})$

$$\Pi(\mathbf{D}) \iff IsPartition(\mathbf{D}) \wedge \mathbf{D} = \langle G_0, G_1, \cdots, G_k \rangle \wedge \forall i, j,\ 0 \leq i, j \leq k,\ i < j - 1 \Rightarrow G_i \not\rightsquigarrow G_j$$

任意のイベント依存グラフ $\mathcal{G}$ について$\Pi(\mathbf{D})$ を満たす分割 $\mathbf{D}$ を持つならば、$\mathcal{G}$ を満たすように Statechart 式通信モデルで動作するオブジェクト集合を構成可能である。 イベント依存グラフから可能なすべての分割の集合を得る関数を $Partitions$ とすれば、

定理 9.1

$$\forall \mathcal{G}.\ \exists \mathbf{D} \in Partitions(\mathcal{G}).\ \Pi(\mathbf{D}) \Rightarrow \exists O.\ \mathcal{G} = Behavior(O, E_{scm}, S, T)$$

Proof 9.1 上の条件を満たす任意のイベント依存グラフに対して、Statechart 式通信モデルでイベント依存グラフの通りに動作する状態遷移図を構成する手順を与えられればよい。

イベント依存グラフ $\mathcal{G} = (F, \rightsquigarrow)$ が$\Pi(\mathbf{D})$ を満たすような分割 $\mathbf{D} = \langle G_0, G_1, \cdots, G_k \rangle$ を持っているものとして、$\mathcal{G}$ 中に現れる任意のオブジェクト o について考える。

$\mathcal{G}$ に現れる o の発火の集合は

$$\{f \in F | Obj(f) = o\}$$

であるが、これらの発火の数を l とすると o の発火は $\mathbf{D}$ の各グループについてたかだか 1 つずつしか含まれていないことから次のような発火のシーケンスを作ることができる。

$$\begin{aligned}\mathbf{F} \quad &= \quad \langle f_0, f_1, \cdots, f_{l-1} \rangle \\ &\text{where} \quad \forall f_n \in \mathbf{F}.\ Obj(f_n) = o \\ &\qquad\quad \forall f_n, f_m \in \mathbf{F}.\ f_n \in G_i \wedge f_m \in G_j \wedge i < j \Rightarrow n < m\end{aligned}$$

さて、o は $\mathcal{G}$ 中に l 回しか発火しないのだから、$\mathcal{G}$ 中に表されている動作に関して o はたかだか $l+1$ 個の状態を持っている。 この $l+1$ 個の状態を $s_0, s_1, \cdots, s_l$として、各々の連続する 2 つの状態についての状態遷移規則を作ればよいことになる。

そして、状態 s_iから状態 s_{i+1}への状態遷移は $\mathcal{G}$ 上では発火 f_iとして表されており、発火 f_iの直接の要因となった発火の集合が $\mathrm{Pre}(f_i)$ で得られるが、条件$\Pi(\mathbf{D})$ より

$$\forall f' \in \mathrm{Pre}(f_i),\ \forall G, G' \in \mathbf{D}.\quad f' \in G \wedge f_i \in G \Rightarrow G \rightsquigarrow G'.$$

また f_iに依存する発火の集合 $\mathrm{Post}(f_i)$ についても同様に条件$\Pi(\mathbf{D})$ より

$$\forall f' \in \mathrm{Post}(f_i),\ \forall G, G' \in \mathbf{D}.\quad f_i \in G \wedge f' \in G' \Rightarrow G \rightsquigarrow G'.$$

つまり、

$$s_i \rightarrow s_{i+1};\quad \bigcup_{f' \in \mathrm{Pre}(f_i)} Ev(f')/Ev(f_i).$$

のような状態遷移規則を作ればよいことになる。

これを $s_0, s_1, \cdots, s_l$のすべての連続する 2 状態について作ってやれば、$\mathcal{G}$ に表されているように動作するオブジェクト o の状態遷移図が構成される。 $\mathcal{G}$ 中に現れるすべてのオブジェクトについて同様に状態遷移図を構成すれば、それらを Statechart 式通信モデルで動作させ、観測して得られたイベント依存グラフは $\mathcal{G}$ に一致する。

End of Proof

また、逆にオブジェクト集合 O を Statechart 式通信モデルで動作させたとき得られるイベント依存グラフは$\Pi(\mathbf{D})$ であるような分割 $\mathbf{D}$ を持つ。つまり

定理 9.2

$$\forall O.\ \mathcal{G} = Behavior(O, E_{scm}, S, T) \Rightarrow \exists \mathbf{D} \in Partitions(\mathcal{G}).\ \Pi(\mathbf{D})$$

Proof 9.2

観測した時間に関する帰納法を使う。 Statechart 式通信モデルでの 1 ステップをイベント依存グラフを分割するときの 1 つのグループに対応させて考えると、

1. オブジェクト集合 O を観測開始から 1 ステップ動作させたとき、そのステップで観測される発火の集合はグループを構成し得るので、これをグループ G_0 とする。

   このときイベント依存グラフはグループ G_0 そのものであるので、分割 $\mathbf{D} = \langle G_0 \rangle$ とすれば明らかに$\Pi(\mathbf{D})$ は成り立つ。

2. オブジェクト集合 O を観測開始から n ステップ動作させたときのイベント依存グラフが分割 $\mathbf{D} = \langle G_0, G_1, \cdots, G_{n-1} \rangle$ を持ち、$\Pi(\mathbf{D})$ が成り立っていると仮定する。

   さらにもう 1 ステップ動作させたとき、新たに観測された発火の集合を G_n とする。 各オブジェクトは直前の n ステップで出力されたイベントのみをトリガとしてたかだか 1 回状態遷移するので、G_n に含まれる発火はすべて G_{n-1} の発火のみに直接依存したものである。つまり、

   $$G_{n-1} \rightsquigarrow G_n \quad \wedge \quad \forall i < n-1.\, G_i \not\rightsquigarrow G_n.$$

   であり、つまりこれは G_n まで含めたイベント依存グラフも分割 $\mathbf{D} = \langle G_0, G_1, \cdots, G_{n-1}, G_n \rangle$ を持ち、これは$\Pi(\mathbf{D})$ を満たしているということになる。

   ゆえに $n+1$ ステップまでの動作についてのイベント依存グラフも条件を満たす。

よってオブジェクト集合 O を Statechart 式通信モデルで動作させたとき、観測開始から有限ステップの間の動作についてイベント依存グラフは条件を満たす。

End of Proof

第 10 章

等価性

本章ではイベント依存グラフを用いた動的モデルのふるまいの等価性について考察する。

10.1 動的モデルのふるまい

同じオブジェクト集合であっても通信モデルが異なれば観測される動作 (つまりイベント依存グラフ) は一般には異なる。つまり

$$Behavior(O, E_{sys}, S, T) \neq Behavior(O, E'_{sys}, S, T)$$

となるのが一般的である。

そこでオブジェクト集合と通信モデルの組から得られるイベント依存グラフについて等価性を定義すれば、着目する期間およびイベント集合に関してオブジェクト集合と通信モデルの組についての観測等価が言えるので、これを動的モデルのふるまいの等価性と定義することができる。

10.2 完全等価の定義

2 つのイベント依存グラフ $\mathcal{G}_1 = (F_1, \rightsquigarrow_1), \mathcal{G}_2 = (F_2, \rightsquigarrow_2)$ が完全等価であることを $\mathcal{G}_1 \equiv \mathcal{G}_2$ と書くとすると

定義 10.1 完全等価

$$\begin{aligned} \mathcal{G}_1 \equiv \mathcal{G}_2 \iff \quad & \exists m : F_1 \to F_2. \quad \forall f_1, f'_1 \in F_1.[f_1 \rightsquigarrow_1 f'_1 \Rightarrow m(f_1) \rightsquigarrow_2 m(f'_1)] \\ & \wedge \quad \forall f_2, f'_2 \in F_2.[f_2 \rightsquigarrow_2 f'_2 \Rightarrow m^{-1}(f_2) \rightsquigarrow_1 m^{-1}(f'_2)] \end{aligned}$$

10.3 部分グラフ

観測によって得られたイベント依存グラフそのものではなく、部分グラフについて等価性を論じたいことがある。

10.3.1 オブジェクトに関する部分グラフ

推移的な依存関係のある 2 つの発火 $f_1 \rightsquigarrow^+ f_2$について、$f_1$から f_2までの依存関係の列で、f_1と f_2の間に現れるすべての発火のオブジェクト集合が O に含まれるとき $f_1 \rightsquigarrow_O f_2$と書く。

定義 10.2

$$f_1 \rightsquigarrow_O f_2 \iff f_1 \rightsquigarrow^+ f_2 \wedge Obj(f_1) \in O \wedge Obj(f_2) \in O \wedge (\forall f_3.\ f_1 \rightsquigarrow^+ f_3 \rightsquigarrow^+ f_2 \Rightarrow Obj(f_3) \notin O).$$

イベント依存グラフ $\mathcal{G} = (F, \rightsquigarrow)$ からオブジェクト集合 O に属さないオブジェクトの発火を取り除いてグラフ $\mathcal{G}' = (F', \rightsquigarrow')$ になることを $SubGraphOnObject(\mathcal{G}, O) = \mathcal{G}'$と書くとすると、

定義 10.3

$$\begin{aligned} &SubGraphOnObject(\mathcal{G}, O) = \mathcal{G}' \\ &\iff \forall f \in F.\ Obj(f) \in O \Rightarrow f \in F' \wedge \forall f_1, f_2 \in F'.\ f_1 \rightsquigarrow_O f_2 \Rightarrow f_1 \rightsquigarrow' f_2 \end{aligned}$$

10.3.2 イベント依存グラフのイベント集合による発火の分離

イベント依存グラフの発火はオブジェクトとイベントインスタンス集合の組みで表されている。 したがってイベント依存グラフ $\mathcal{G} = (F, \rightsquigarrow)$ とイベント集合 E が与えられたとき、F 中の発火 $f = (o, \overline{E})$ と E から作られるイベントインスタンス集合$\overline{E_0} = \{(e, n) | e \in E, n \in N\}$ について

(1). $\overline{E} \subseteq \overline{E_0}$であるもの、

(2). $\overline{E} \not\subseteq \overline{E_0} \wedge \overline{E} \cap \overline{E_0} \neq \emptyset$ であるもの、

(3). $\overline{E} \cap \overline{E_0} = \emptyset$ であるもの

が考えられる。 そこで (2) のような発火から$\overline{E_0}$に属さないようなイベントインスタンスを除去して (1) のような発火 f'にしたとき $f' = SubFiring(f, E)$ と書くとする。

定義 10.4

$$SubFiring(f, E) = f' \iff Ei(f') = Ei(f) \cap \{(e, n) | e \in E, n \in N\} \wedge Obj(f') = Obj(f)$$

10.3.3 イベントに関する部分グラフ

発火 f_1, f_2 とイベント集合 E について次のような関係 $\rightsquigarrow_E$ を定義する。

定義 10.5

$$f_1 \rightsquigarrow_E f_2 \iff f_1 \rightsquigarrow^+ f_2 \wedge Ev(f_1) \subseteq E \wedge Ev(f_2) \subseteq E \wedge \forall f_3.\ f_1 \rightsquigarrow^+ f_3 \rightsquigarrow^+ f_2 \Rightarrow Ev(f_3) \cap E = \emptyset$$

イベント依存グラフ $\mathcal{G} = (F, \rightsquigarrow)$ からイベント集合 E に属さないイベントに関する発火を取り除いてグラフ $\mathcal{G}' = (F', \rightsquigarrow')$ になることを $SubGraphOnEvent(\mathcal{G}, E) = \mathcal{G}'$ と書くとすると、

定義 10.6

$$SubGraphOnEvent(\mathcal{G}, E) = \mathcal{G}' \iff$$
$$(\forall f \in F.\ \ Ev(f) \cap E \neq \emptyset \Rightarrow SubFire(f, E) \in F')$$
$$\wedge (\forall f_1', f_2' \in F',\ \ \exists f_1, f_2 \in F.\ \ SubFire(f_1, E) = f_1' \wedge SubFire(f_2, E) = f_2' \wedge f_1 \rightsquigarrow_E f_2 \Rightarrow f_1' \rightsquigarrow' f_2')$$

10.4 部分等価の定義

イベント依存グラフの部分グラフの定義からイベント依存グラフの部分等価を以下のように定義する。

定義 10.7 オブジェクトに関する部分等価

2 つのイベント依存グラフ $\mathcal{G}_1, \mathcal{G}_2$ とオブジェクト集合 O について、O に関する $\mathcal{G}_1$ と $\mathcal{G}_2$ の部分グラフが完全等価ならば、$\mathcal{G}_1$ と $\mathcal{G}_2$ は O について部分等価である。

$$\mathcal{G}_1 \equiv_O \mathcal{G}_2 \iff SubGraphOnObject(\mathcal{G}_1, O) \equiv SubGraphOnObject(\mathcal{G}_2, O)$$

同様にイベントについても

定義 10.8 イベントに関する部分等価

2 つのイベント依存グラフ $\mathcal{G}_1, \mathcal{G}_2$ とイベント集合 E について、E に関する $\mathcal{G}_1$ と $\mathcal{G}_2$ の部分グラフが完全等価ならば、$\mathcal{G}_1$ と $\mathcal{G}_2$ は E について部分等価である。

$$\mathcal{G}_1 \equiv_E \mathcal{G}_2 \iff SubGraphOnEvent(\mathcal{G}_1, E) \equiv SubGraphOnEvent(\mathcal{G}_2, E)$$

定義 10.9 イベントとオブジェクトに関する部分等価

2 つのイベント依存グラフ $\mathcal{G}_1, \mathcal{G}_2$、オブジェクト集合 O、イベント集合 E について、$\mathcal{G}_1 \equiv_E \mathcal{G}_1' \wedge \mathcal{G}_2 \equiv_E \mathcal{G}_2'$ を満たすようなグラフ $\mathcal{G}_1', \mathcal{G}_2'$ が存在し、かつ、$\mathcal{G}_1' \equiv_O \mathcal{G}_2'$ であるとき、$\mathcal{G}_1$ と $\mathcal{G}_2$ は E と O について等価であるといい、$\mathcal{G}_1 \equiv_{O,E} \mathcal{G}_2$ と書く。

$$\mathcal{G}_1 \equiv_{O,E} \mathcal{G}_2 \iff \exists \mathcal{G}_1', \mathcal{G}_2'.\ \ \mathcal{G}_1 \equiv_E \mathcal{G}_1' \wedge \mathcal{G}_2 \equiv_E \mathcal{G}_2' \wedge \mathcal{G}_1' \equiv_O \mathcal{G}_2'$$

第 11 章

Statechart 式通信モデルと非同期通信モデル

本章では、前章で定義した動的モデルのふるまいの等価性に基づいて Statechart 式通信モデルと個別実行モデルとの相互シミュレートを研究する。 なお、本章の例で用いるイベント依存グラフについて、図面上ではイベントのインスタンスを区別する整数は省略し、イベント名のみを用いて発火を表すものとする。

11.1 異なるイベントシステムの相互シミュレート

あるイベントシステム E_{sys}でのオブジェクト集合 O の動作を、別のイベントシステム E'_{sys}でシミュレートさせることを考える。 ただし、O を全く変更せずにシミュレートすることは一般には難しいので、 イベントシステムの違いを吸収するように O を変更して O'として E'_{sys}で動作させ、双方のイベント依存グラフが O について等価であるようにする。 つまり、任意の O に対して

$$Behavior(O, E_{sys}, S, T) \equiv_O Behavior(O', E'_{sys}, S, T)$$

であるような O'を構成する方法を考える。

11.2 個別実行モデルの Statechart 式通信モデルでのシミュレーション

あるオブジェクト集合の個別実行モデルに基づく任意のイベントシステムにおけるイベント依存グラフに対して、そのオブジェクト集合への特定の変更によって、Statechart 式通信モデルで元のイベント依存グラフを実現させることができる。 つまり、次のような定理が成り立つ。

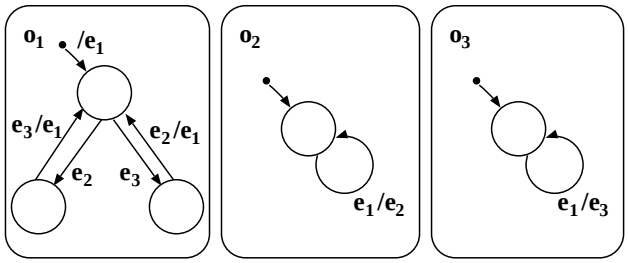

Figure 11.1: 書き換え前図

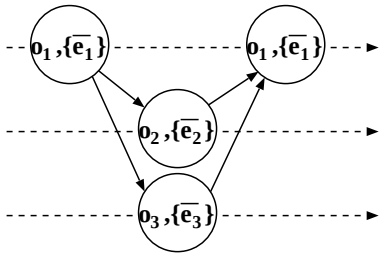

Figure 11.2: 書き換え前のイベント依存グラフ

定理 11.1

任意のオブジェクト集合 O_1 に対して

$$\forall E_{sys} \in E_{ind}.\ \ Behavior(O_1, E_{sys}, S, T) \equiv_{O_1} Behavior(O_2, E_{scm}, S, T).$$

を満たすように O_2 が構成可能である。

この定理の一般の証明はここでは示さないが、任意のオブジェクト集合 O_1 に対して定理を満たすような O_2 を構成する方法とその具体例を示すことで定理の証明に換えることとする。

さて、個別実行モデルで複数の発火が同時に起こらないことから、Statechart 式通信モデルの 1 ステップに個別実行モデルでの 1 回の発火を割り当てるように各オブジェクトの発火を制御してやれば個別実行モデルでの動作をシミュレートできる。 このために各オブジェクトの発火で出力されるイベントを他のオブジェクトには届かないように回収しておき、それらのイベントを複数の発火が同時に起こらないようにシリアライズして送るためのシリアライザオブジェクトを作成する。

0. 前提

   個別実行モデルで動作しているオブジェクト集合 $O_1 = \{o_1, o_2, \cdots, o_m\}$ があり、それらの状態遷移図中で使用されるイベントの集合は $E_1 = \{e_1, e_2, \cdots, e_n\}$ であるとする。

   ここでは書き換えの例として図 11.1のような並行動作する 3 つのオブジェクトを考える。これらのオブジェクトを個別実行モデルで動作させたときのイベント依存グラフは図 11.2のようになっている。

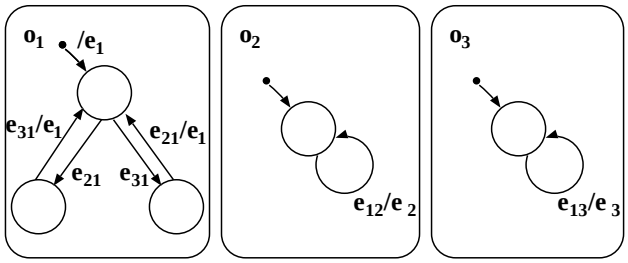

Figure 11.3: 入力イベント置き換え後

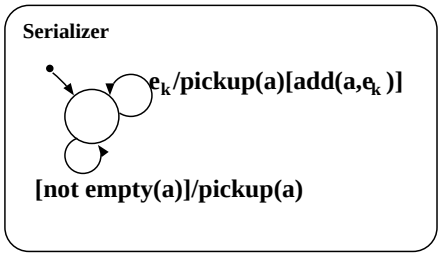

Figure 11.4: シリアライザ

1. 入力イベントの置き換え

   各オブジェクトの状態遷移図にあるすべての状態遷移規則の入力イベントの部分を置き換える。オブジェクト o_j の遷移規則の入力イベント e_i を e_{ij} に置き換える。ここで $\{e_{ij}|1 \leq i \leq n, 1 \leq j \leq m\}$ は新たに追加したイベントの集合である。

   図 11.1に対して入力イベントの置き換えを行うと図 11.3のようになる。

2. シリアライザオブジェクトの作成

   与えられたオブジェクト集合 O_1 の全体と並行に動作するようにシリアライザオブジェクト $Serializer$ を作成する。

   $Serializer$ はイベントを蓄えるバッファを持ち、 他のオブジェクトの発火 f に対しては、与えられたイベント依存グラフから f に依存する発火の集合 $\mathrm{Post}(f)$ が得られるので、イベントの集合 $\{e_{ij}|\exists f' \in \mathrm{Post}(f) \wedge Obj(f') = o_j\}$ をイベントバッファに追加する一方で、バッファ中のイベントを 1 ステップにつき 1 イベントずつ出力していく。

   名前の付け替え後のイベントを 1 ステップに 1 つずつ出力していくことで、各ステップに $Serializer$ 以外のオブジェクトはどれか 1 つしか発火しないことになり、また、各オブジェクトの発火を $Serializer$ のバッファ中にため込んでいくことで、Statechart 式通信モデルのイベントの有効期限に制限されずに与えられたイベント依存グラフの動作を実現できる。

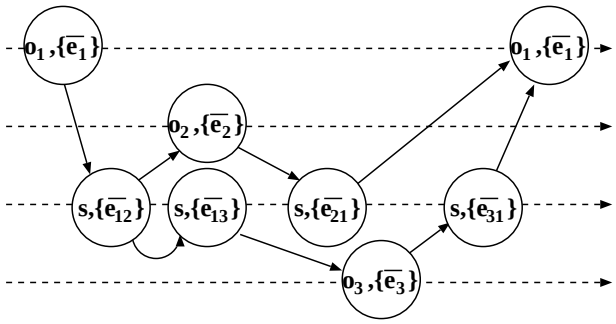

Figure 11.5: 書き換え後のイベント依存グラフ

先程の例に対してシリアライザオブジェクトを作成すると図 11.4のようになる。 ここでは $Serializer$ には集合構造を持つ属性 a があり、a について以下のような操作、関数を持つものとして記述した。

add(a, e_k) イベント e_kに対してイベント依存グラフから依存関係を持ち得る発火を調べて、これに対して出力すべきイベント集合を属性 a に追加する操作

pickup(a) 与えられたイベントシステムにしたがって次に出力すべきイベントを属性 a から抜き出す関数

empty(a) 属性 a が空であるかどうかを調べる述語

なお、ここではわかりやすくするために単純化して状態遷移図を書いているが、 実際には e_kをトリガとする遷移規則は実際には e_1, e_2, e_3の 3 つのイベントに対してそれぞれ遷移規則があり、また、それらの遷移がいずれも成り立たない場合にのみ [**not empty**(a)] で始まる遷移を見るように優先度が付けられているものとする。

3. 結果

以上のようにして作成すると

オブジェクト集合　$O_2 = O_1 \cup \{Serializer\},$

イベント集合　　$E_2 = E_1 \cup \{e_{ij} | 1 \leq i \leq n, 1 \leq j \leq m\}.$

のようなものとなる。書き換え後の状態遷移図を用いて (O_2ではなく)O_1のオブジェクト集合に関してのイベント依存グラフを考えると、$Serializer$ 経由での依存関係が着目するオブジェクト集合を制限することで直接の依存関係として現れる。 書き換え後の O_2に関するイベント依存グラフは図 11.5 のようになっている。オブジェクト名 S と書いてあるのがシリアライザである。 このイベント依存グラフから O_1についての部分グラフを求めると書き換え前の図 11.2と一致する。つまり、

$$\forall E_{sys} \in E_{ind}.\ \ Behavior(O_1, E_{sys}, S, T) \equiv_{O_1} Behavior(O_2, E_{scm}, S, T).$$

である。

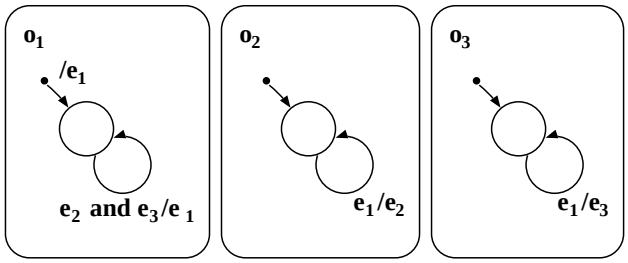

Figure 11.6: 書き換え前

この節ではO_1の個別実行モデルで動作を Statechart 式通信モデルでシミュレートするものとして説明してきたが、実際には与えられたイベント依存グラフにしたがって動作するようにO_1の書き換えをしているので、任意のイベント依存グラフについて Statechart 式通信モデルでシミュレートできることになる。

11.3 Statechart 式通信モデルの個別実行モデルでのシミュレーション

あるオブジェクト集合の Statechart 式通信モデルにおけるイベント依存グラフに対して、そのオブジェクト集合への特定の変更によって、個別実行モデルに基づく任意のイベントシステムで元のイベント依存グラフを実現させることができる。 つまり、次のような定理が成り立つ。

定理 11.2
任意のオブジェクト集合O_1およびO_1中のオブジェクトが用いるイベントの集合E_1に対して

$$\forall E_{sys} \in E_{ind}.\ \ Behavior(O_1, E_{scm}, S, T) \equiv_{O_1,E_1} Behavior(O_2, E_{sys}, S, T).$$

を満たすようにO_2が構成可能である。

この定理の一般の証明はここでは示さないが、任意のオブジェクト集合O_1に対して定理を満たすようなO_2を構成する方法とその具体例を示すことで定理の証明に換えることとする。

Statechart 式通信モデルでは 1 ステップの間に各オブジェクトはたかだか 1 回しか発火しないことから、各オブジェクトの発火で出力されるイベントを回収しておき、それらのイベントによって各オブジェクトがたかだか 1 回ずつ発火するように制御するシンクロナイザオブジェクトを作成する。

0. 前提

   Statechart 式通信モデルで動作しているオブジェクト集合$O_1 = \{o_1, o_2, \cdots, o_m\}$があり、それらの状態遷移図中で使用されるイベントの集合は$E_1 = \{e_1, e_2, \cdots, e_n\}$であるとする。

   ここでは書き換えの例として図 11.6のような並行動作する 3 つのオブジェクトを考える。これらのオブジェクトを Statechart 式通信モデルで動作させたときのイベント依存グラフは図 11.7のようになっている。

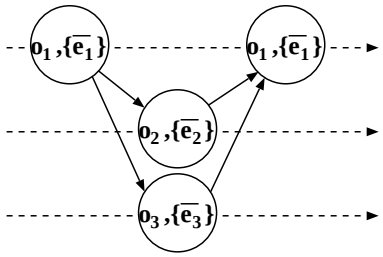

Figure 11.7: 書き換え前のイベント依存グラフ

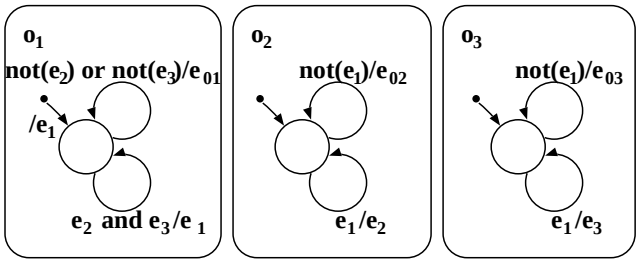

Figure 11.8: 書き換え中

1. 出力イベントの追加と置き換え

   まず、Statechart 式通信モデルで動作させたときに各オブジェクトについてイベントを出力しないステップがあるのならば、そのステップでもイベントを出力しないことを意味する特別なイベントを出力するように状態遷移規則を追加/変更する。オブジェクト o_jであればイベント e_{0j}を出力するようにする。

   ここまで書き換えると図 11.8のようになる。

   さらに、もともと出力している $e_1, \cdots, e_n$のイベントについてどのオブジェクトから出力されたかを区別できるように遷移規則の出力イベントの部分を書き換える。オブジェクト o_jの遷移規則の出力

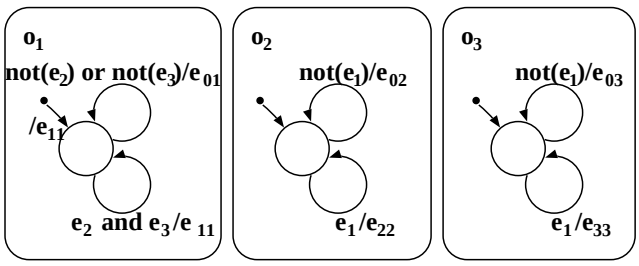

Figure 11.9: 書き換え中

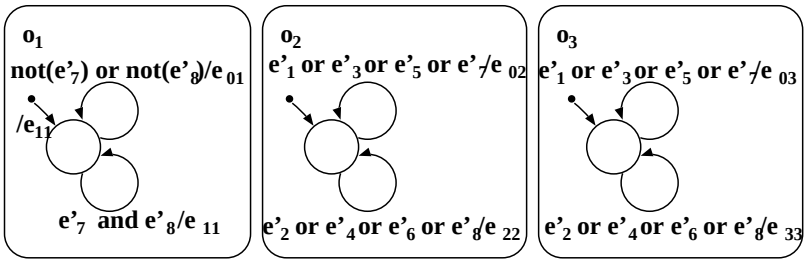

Figure 11.10: 入出力イベント書き換え後

イベント e_iを e_{ij}に置き換える。

ここまで書き換えると図 11.9のようになる。

2. 入力イベントの置き換え

書き換え前の n 種類のイベントについて、各イベントが出力されているかどうかは 2^n個のパターンが有り得る。 その全パターンに対応するように新たなイベント $e'_k,\ 1 \leq k \leq 2^n$を作成し、各オブジェクトの状態遷移規則の入力イベントの部分はすべてこの新たなイベントで置き換える。

図 11.6〜11.9までの例で

$$
\begin{aligned}
e'_1 &= \text{not}(e_1) \text{ and not}(e_2) \text{ and not}(e_3) \\
e'_2 &= e_1 \text{ and not}(e_2) \text{ and not}(e_3) \\
e'_3 &= \text{not}(e_1) \text{ and } e_2 \text{ and not}(e_3) \\
e'_4 &= e_1 \text{ and } e_2 \text{ and not}(e_3) \\
e'_5 &= \text{not}(e_1) \text{ and not}(e_2) \text{ and } e_3 \\
e'_6 &= e_1 \text{ and not}(e_2) \text{ and } e_3 \\
e'_7 &= \text{not}(e_1) \text{ and } e_2 \text{ and } e_3 \\
e'_8 &= e_1 \text{ and } e_2 \text{ and } e_3
\end{aligned}
$$

のようにおいて入力イベントを書き換えると図 11.10のようになる。

3. シンクロナイザオブジェクトの作成

与えられたオブジェクト集合O_1の全体と並行に動作するようにシンクロナイザオブジェクト $Synchronizer$を作成する。$Synchronizer$ の動作は、各オブジェクトから $e_{01}, e_{11}, \cdots, e_{n1}$のうちの 1 つ、$e_{02}, e_{12}, \cdots, e_{n2}$のうちの 1 つ、$\cdots$、$e_{0m}, e_{1m}, \cdots, e_{nm}$のうちの 1 つを任意の順番で受け取り、もらったイベントのパターンに応じてイベント $e'_k,\ 1 \leq k \leq 2^n$を出力する。

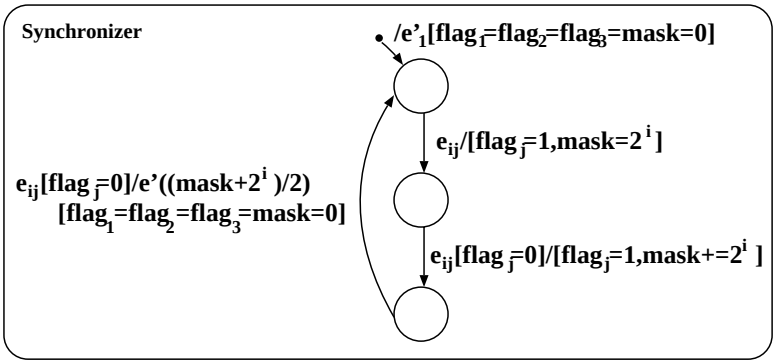

Figure 11.11: シンクロナイザ

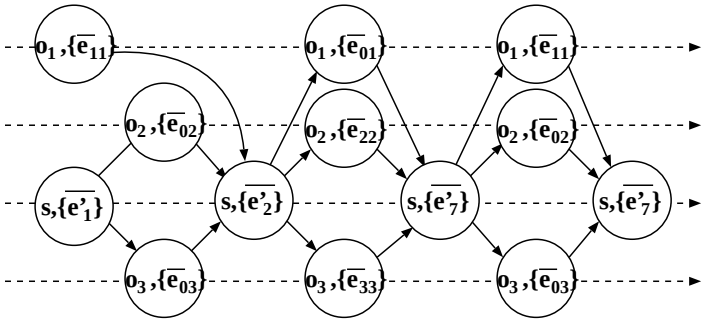

Figure 11.12: 書き換え後のイベント依存グラフ

先程の例に対してシンクロナイザオブジェクトを作成すると図 11.11のようになる。ここではシンクロナイザオブジェクトが受ける必要のあるすべてのイベント列を遷移規則に直すと非常に複雑になるため、図面の上では簡略化するために $e_{01}, e_{11}, e_{21}, e_{31}$の 1 つを受け取ったことを表す属性 $flag1$、$e_{02}, e_{12}, e_{22}, e_{32}$の 1 つを受け取ったことを表す属性 $flag2$、$e_{03}, e_{13}, e_{23}, e_{33}$の 1 つを受け取ったことを表す属性 $flag3$、および、これらのイベントの組合せから $e'_1, \cdots, e'_8$のうちの 1 つへの変換を表す属性 $mask$ を考え、かつ、属性評価式には C ライクな式が書けるものとして記述した。

4. 結果

以上のようにして作成すると、

オブジェクト集合　$O_2 = O_1 \cup \{Synchronizer\}$,

イベント集合　　$E_2 = E_1 \cup \{e_{ij} | 0 \leq i \leq n, 1 \leq j \leq m\} \cup \{e'_k | 1 \leq k \leq 2^n\}$.

のようなものとなる。 書き換え後の O_2に関するイベント依存グラフは図 11.12のようになっている。オブジェクト名 S と書いてあるのがシンクロナイザである。 ここからイベント e_{01}, e_{02}, e_{03}以外のイベントについて部分グラフを取ると図 11.13のようになり、 O_1について部分グラフを取ると図 11.14のようになる。これは図 11.7と完全等価になっている。 つまり、$E' = E_2 - \{e_{01}, e_{02}, e_{03}\}$ とした

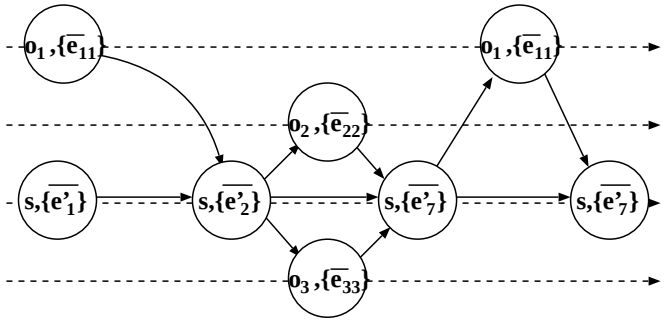

Figure 11.13: e01,e02,e03 以外のイベントに関する部分グラフ

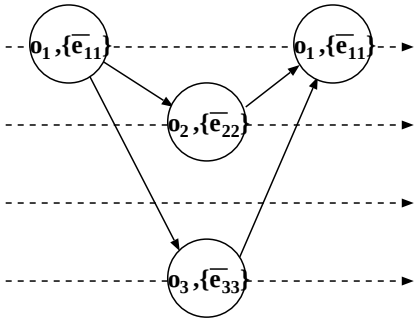

Figure 11.14: O_1についての部分グラフ

とき

$$\forall E_{sys} \in E_{ind}.\ \ Behavior(O_1, E_{scm}, S, T) \equiv_{O_1, E_1} Behavior(O_2, E_{sys}, S, T).$$

である。

第 12 章

まとめ

本論文ではオブジェクト指向方法論の動的モデルを形式的に記述できる仕様記述モデル ObTS を提案し、Statechart 式通信モデルに基づく操作的セマンティクスを与えた。ObTS のための記述言語として ObCL を提案し、その記述支援およびシミュレーションのための環境を構築した。

次に動的モデルにおける通信モデルの形式化としてイベント依存グラフを提案し、制約されたイベント依存グラフの Statechart 式通信モデルでの実現可能性を示した。これにより個々のオブジェクトのふるまいを状態遷移図で、全体の動作をイベント依存グラフで記述することで、プロトタイプ実行可能であるという点で有用な仕様記述となることを示した。さらにイベント依存グラフの上で動的モデルのふるまいの等価性を定義し、その等価性のもとで動作を調整するオブジェクトの追加と既存オブジェクトのイベント名付け替えによって、任意のイベント依存グラフを Statechart 式通信モデルで実現可能であることを示した。また、逆に Statechart 式通信モデルで実現可能なイベント依存グラフに対しても、同様の書き換えによって通常の非同期通信モデルでの実現が可能であることを示した。

ObTS のように Statechart 式通信モデルを採用する仕様記述モデルは、非同期通信を行うような記述対象に対しては直接的でないと考えられがちだが、Statechart 式通信モデルの意味付けを明らかにすることでそのような記述対象に対しても充分に有効であることがわかる。また、イベント依存グラフでは外から観測されるオブジェクトのふるまいを示しているので、オブジェクトの動的ふるまいに関する外部仕様としての意味があり、これに対して Statechart 式通信モデルの性質を示したことで外部仕様に対する Statechart 式通信モデルの実現可能性を与えている。

今後の課題としては ObTS および ObCL について規模の大きい事例に適用することで、その有効性を確認するとともに、ObCL 環境でのテストや検証について研究する。また、個々のオブジェクトに関する状態遷移図とイベント依存グラフを用いた仕様記述についての研究を行う。

謝辞

本研究を行なうに当たり、終始御指導を賜わった片山卓也教授に深謝致します。
また、本論文をまとめるに当たって御協力いただいた片山研究室の諸兄に厚く御礼申し上げます。

参考文献

[1] D.Harel, A.Pnueli, J.P.Schmid and R.Sherman, *On the Formal Semantics of Statecharts*, Proc of 2nd IEEE Symposium on Logic in Computer Science, 54-64,1987.

[2] Michael von der Beeck, *A Comparison of Statecharts Variants*, LNCS 863, Formal Techniques in Real-Time and Fault-Tolerant Systems, 128-148, 1994.

[3] J.Rumbaugh, M.Blaha, W.Premerlai, F.Eddy, W.Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall International,Inc.,1991

[4] D.Coleman, F.Hayes and S.Bear, *Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design*, IEEE Trans. on Software Engineering, Vol. 18, No.1, 1992.

[5] D.Gangopadhyay, S.Mitra, *ObjChart: Tangible Specification of Reactive Object Behavior*, E-COOP'93.

[6] D.Harel, E.Gery, *Executable Object Modeling with Statecharts*, ICSE-18, 246-, 1996

[7] Statemate MAGNUM User Guide, i-Logix.

本研究に関する発表論文

[1] 伊藤 恵: オブジェクト指向方法論のための動的モデル *ObTS*, 北陸先端科学技術大学院大学 情報科学研究科 修士論文,1995.

[2] 伊藤 恵, 片山 卓也: オブジェクト指向方法論のための動的モデルにおける *Statechart* 式通信モデル, ソフトウェア科学会第 13 回大会論文集, pp.445-448, 1996.

[3] 伊藤 恵, 片山 卓也: オブジェクト指向方法論のための動的モデル *ObTS*, コンピュータソフトウェア vol.14 No.2, pp.22-37, March 1997.