ソフトウェア工学 131－6
（2001．6．1）

メタ階層化による図式モデルの形式的記述

小飼 敬　　上田 賀一

茨城大学 工学部 情報工学科

〒316-8511 茨城県日立市中成沢町 4-12-1

ソフトウェア開発においてソフトウェアの仕様を矛盾や曖昧さがないように記述するためには，仕様を形式化して検証するというアプローチが有効である．しかし仕様の規模が大きくなると，形式化する手間も多くなってくるため作業が困難となる．本研究では，ソフトウェア仕様で利用される図式モデルに焦点をあて，図式モデルを記述するための制約をメタモデルとして定義した．これによって，仕様記述者はメタモデルの制約に従ってモデルを形式化できるため，この作業が容易になる．また，メタモデルに対しては形式的記述の際の指針を与えることで，ある程度手間を軽減することができる．

Formal Specifications of Diagrammatic Models using Meta Hierarchy

Kei Kogai　　Yoshikazu Ueda

Ibaraki University

4-12-1 Naka-Narusawa, Hitachi, Ibaraki, 316-8511 Japan

A software specification is formalized so that it is guaranteed to be described without a contradiction and a ambiguity. In large scale software, however, it is difficult to formalize specifications. This work focuses on diagrammatic models using software specifications, and defines the meta models that have a semantic description of a diagrammatic model. So, it is enable that a model is described using its meta model, consequently the specifications is formalized simply. For meta models, this work proposes a guideline for formalizing meta models to describe formal specifications easily.

1　はじめに

ソフトウェア開発においてソフトウェアの仕様を矛盾や曖昧さがないように記述することは重要であり，これを検証するために仕様を形式化するというアプローチは有効である．しかしシステムの大規模化にともなって，その仕様も大きくなり，形式化する手間も多くなってくる．そのような状況で何もないところからの形式化は困難であるため，本研究では仕様として扱われる図式モデルに対して形式的仕様をメタ階層化し，その記述のための指針を与えることで形式的仕様の記述を容易に行えるようにする．

本報告では，まず形式的仕様のメタ階層化について述べ，その後に様々な種類の図式モデルのメタモデルを記述する際の指針について述べる．

2 図式モデルのメタ階層化

何も形式化されいない状態から図式モデルの形式的仕様を記述することは困難である．これを解決するために対象とした図式モデルの記述要素を定義したモデル，つまりメタモデルをあらかじめ形式的仕様として与え，メタモデルを用いることで容易に図式モデルの形式的仕様を記述することを目指す．

しかし，図式モデルには様々な種類があるため，これらに対応したメタモデルも様々な種類が存在する．このような様々な種類のメタモデルの形式的仕様を記述することのできる記述モデルが必要となる．そこで，本研究ではこの記述モデルとしてメタモデルのメタモデル，つまりメタメタモデルを用意し，更にメタモデルの記述に対する指針を与えることで，メタモデルの形式的仕様が比較的容易に行えるようにする．

また，メタモデルを用いて記述した図式モデルのことをベースモデルと呼ぶことにする．これで図式モデルは，メタメタモデル，メタモデル，ベースモデルの 3 つのレベルに階層化され表現される．

2.1 メタメタモデルの定義

メタメタモデルには，様々なメタモデルを記述できるように実体関連モデルを考える．更にモデルを包含できるようにフィールドといった要素を加える．またフィールドは要素を包含できる実体として扱う．

メタメタモデルを実体関連モデルで表すと図 ?? のようになる．図中の要素 ENTITY, RELATIONSHIP, FIELD はそれぞれエンティティ，リレーションシップ，フィールドに対応する．

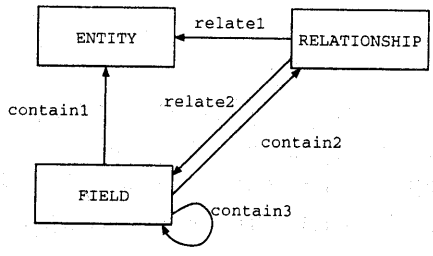

図 1: メタメタモデル

3 Z による図式モデルの記述

本節では図式モデルの形式的仕様をメタ階層化するにあたって必要な概念を順に Z 言語 [?][?] で記述していく．そして，メタモデルの仕様を記述するための指針を示してから，ペトリネットモデルを例として，メタ階層化された図式モデルの形式的仕様の記述方法について述べる．

3.1 基礎となるモデルの仕様

前節で述べたように，本研究では図式モデルの仕様は実体関連モデルで表す．このためこれらのモデルの仕様を表現するのには，ノード・アークモデルが適している．従って，まず基礎概念としてノード・アークモデルの仕様を形式化する．図式モデルを構成する要素は全て実体が存在するため，これに対応したノードを表すための型が必要となる．この型を $NODE$ とする．

$$[NODE]$$

次にノードがアークによって関連付けられている状態の表現について考える．アークは常に 2 つノード間のみを関連付けているとして，この関連付けを関数 $relate$ を用いて次のように表す．

$$(node1, node2) = relate(arc)$$

ここで，式中の $node1$, $node2$, arc は集合 $NODE$ の要素である．これによって，定義域はノードを関連付けているアークであり，値域は関連元のノードと関連先のノードの対を表している．つまりアーク arc はノード $node1$ からノード $node2$ へと関連付けている．

そして，ノード・アークモデルをスキーマ $NAModel$ として次のように定義する．このスキーマの述部には，アークが存在する時は必ず 2 つのノードを関連付けているいることを表している．

$$
\begin{array}{|l}
\hline
NAModel \\
Arc : \mathbb{P}\, NODE \\
Node : \mathbb{P}\, NODE \\
relate : NODE \nrightarrow (NODE \times NODE) \\
\hline
Arc = \mathrm{dom}\, relate \\
(Node \times Node) = \mathrm{ran}\, relate \\
\forall a : Arc \bullet \\
\quad \exists n1, n2 : Node \bullet (n1, n2) = relate(a) \\
\hline
\end{array}
$$

スキーマ *NAModel* により，ノード・アークモデルの状態を表すことができるが，図式モデルの中にはモデルの構成要素が生成または削除されることによってモデル全体の状態の変化を表すモデルがある．例えば，ペトリネットモデルはトークンという構成要素が変化することでモデルの状態が変化する．このような図式モデルを考慮して，モデルの各構成要素に対する生成と削除を表すスキーマを定義する．

ノード・アークモデルの場合，構成要素としてノードとアークが存在するため，この2つの要素に対する生成と削除の仕様を操作スキーマとして定義する．ノードの生成では，生成されるノードが既に存在するノードの集合 *Node* の要素でないことが事前条件であり，生成後は *Node* の要素となることが事後条件である．

CreateNode

$\Delta NAModel$
$node? : NODE$

$node? \notin Node$
$Node' = Node \cup \{node?\}$

一方，ノードの削除では，削除されるノードが既に存在するノードの集合 *Node* の要素であることが事前条件であり，削除後は *Node* から削除されることが事後条件である．

DeleteNode

$\Delta NAModel$
$node? : NODE$

$node? \in Node$
$Node' = Node \setminus \{node?\}$

アークはノードを関連付けることが生成目的であるため，まず関連付けの対象となる2つノードが存在していなければならない．従って，アーク作成時の事前条件は，ノードの場合の条件に加えて，関連付けるアークが既に存在することである．また事後条件はノードの場合に加えて，これらのノードを関連付けることである．

CreateArc

$\Delta NAModel$
$arc? : NODE$
$node1? : NODE$
$node2? : NODE$

$arc? \notin Arc$
$node1? \in Node$
$node2? \in Node$
$relate' = relate \cup \{arc? \mapsto (node1?, node2?)\}$

そして，アークの削除は，削除されるアークが既に存在するアークの集合 *Arc* の要素であることが事前条件であり，削除後は関連付けを表す関数 *relate* から削除されたアークの関連付けを取り除かれていることが事後条件である．

DeleteArc

$\Delta NAModel$
$arc? : NODE$

$arc? \in Arc$
$relate' = \{arc?\} \lhd relate$

3.2 メタ階層化されたモデル間の関係

図式モデルのベースモデルの記述はそのメタモデルの定義に従って記述されることになる．従って図式モデルの仕様を記述する際に，メタモデルに関する定義を参照できる必要がある．そこで，モデルの各構成要素と自身のメタモデルとの相互関係を表すスキーマを定義する．メタ階層化されたモデル間の関係をスキーマ *MetaHierarchy* として次のように定義する．ベースモデルからそのメタモデルへの参照は関数 *meta* で表し，メタモデルとそのメタモデルから生成されたベースモデルの集合との関係を関数 *base* で表している．

MetaHierarchy

$meta : NODE \nrightarrow NODE$
$base : NODE \nrightarrow \mathbb{P}\, NODE$
$BaseModel : \mathbb{P}\, NODE$
$MetaModel : \mathbb{P}\, NODE$

$BaseModel = \mathrm{dom}\, meta$
$MetaModel = \mathrm{ran}\, meta$
$MetaModel = \mathrm{dom}\, base$
$\mathbb{P}\, BaseModel = \mathrm{ran}\, base$
$\forall basemodel : NODE \bullet$
$\quad \exists metamodel : NODE \bullet$
$\quad\quad metamodel = meta(basemodel)$
$\forall basemodel, metamodel : NODE \bullet$
$\quad metamodel = meta(basemodel) \Leftrightarrow$
$\quad basemodel \in base(metamodel)$

3.3 記述に対する指針

前節までで定義したノード・アークモデルとメタ階層化されたモデル間の関係を使うと，様々な図式モデルのメタモデルの形式的仕様を記述できるわけだが，これだけで記述するのは困難である．そこでどの図式モデルのメタモデルにも共通して利用できる，スキーマの変数宣言や述語に対する指針を示す．

メタモデルの状態に関するスキーマ

まずメタモデルの状態スキーマに関する記述の指針について述べる．ここには構成要素とそれらの関連の宣言やここから生成されるモデルに対する記述などがある．詳細は以下のようになる．

- メタモデルを構成する要素となる変数の宣言
- メタモデルを使って記述するベースモデルでの要素の集合を表す変数の宣言
- メタモデルを使って記述するベースモデルでの関連を表す関数の宣言
- 構成要素とそれのベースモデルとの関係の *base* による記述
- ベースモデルで関連を表す関数の定義域と値域に関する記述
- メタモデルの構成要素とそのメタモデルの関係に関する記述
- ベースモデルにおける関連付けはメタモデルの定義に従っていることの記述
- ERF モデルで表されたメタモデルの構成要素間の関連に関する記述

ノードタイプの生成/削除に関するスキーマ

次にベースモデルの構成要素生成に関するスキーマについて考える．本研究では，基盤となる表現形式にノード・アークモデルを用いているため，そこから記述される図式モデルの構成要素はノードタイプのものとアークタイプのものの 2 種類が存在する．このため，要素の生成/削除の操作に関するスキーマの記述の指針も 2 つの種類が考えられる．まず，ノードタイプの構成要素に関するスキーマ記述の指針を挙げる．

- 生成されるモデル構成要素の変数宣言
- 生成される構成要素はまだベースモデルとして存在しないという記述（事前条件）
- ベースモデルを表す集合に生成された構成要素が含まれるという記述（事後条件）

アークタイプの生成/削除に関するスキーマ

次に，アークタイプに関する記述の指針を挙げる．アークタイプの構成要素はノードタイプの構成要素を関連付けるために存在するので，関連付けの対象となる構成要素が既に存在している必要がある．この制約がノードタイプの事前条件と事後条件に加えられる．

- 生成される構成要素と関連付けの対象となる構成要素の変数宣言
- 生成される構成要素はまだベースモデルとして存在しないという記述（事前条件）
- 関連付けの対象となる構成要素は既に存在しているという記述（事前条件）
- ベースモデルを表す集合に生成された構成要素が含まれるという記述（事後条件）
- 関連付けを表す関数にこの関連付けに対応した写像が追加されてたという記述（事後条件）

3.4 メタメタモデルの形式化

図 **??**に示すように，メタメタモデル自身は実体関連モデルで表せるため，先に示したノード・アークモデルと記述の指針と同様に仕様を記述することができる．ここではまずメタメタモデルを記述するのに必要となる ERF モデルの仕様を記述する．ERF モデルを表すスキーマを *ERFModel* として定義する．ERF の基本要素となるエンティティ，リレーションシップ，フィールドの集合をそれぞれ *EntitySet*, *RelationshipSet*, *FieldSet* として宣言する．そして，リレーションシップがエンティティを関連付けている状態を関数 *relateEtoE* として表す．これらの関数の制約はこのスキーマの述部の前半に記述してある．また，フィールドによる構成要素の包含関係を表すために関数 *contain* を導入している．

ERFModel

$NAModel$
$MetaHierarchy$
$EntitySet : \mathbb{P}\, NODE$
$RelationshipSet : \mathbb{P}\, NODE$
$FieldSet : \mathbb{P}\, NODE$
$Contain : \mathbb{P}\, NODE$
$relateEtoE : NODE \pfun (NODE \times NODE)$
$contain : NODE \pfun (NODE \times NODE)$

$EntitySet \cap RelationshipSet = \varnothing$
$FieldSet \subseteq EntitySet$
$Contain \subseteq Arc$
$RelationshipSet = \mathrm{dom}\, relateEtoE$
$EntitySet \times EntitySet = \mathrm{ran}\, relateEtoE$
$Contain = \mathrm{dom}\, contain$
$(FieldSet \times (EntitySet \cup RelationshipSet))$
$\quad = \mathrm{ran}\, contain$
$\forall r : RelationshipSet;\ e1, e2 : EntitySet \bullet$
$\quad relateEtoE(r) = (e1, e2) \Leftrightarrow$
$\quad\quad (\exists a : Arc \bullet relate(a) = (e1, r)) \land$
$\quad\quad (\exists a : Arc \bullet relate(a) = (r, e2))$
$\forall r : RelationshipSet \bullet$
$\quad \exists e1, e2 : EntitySet \bullet$
$\quad\quad (e1, e2) = relateEtoE(r)$
$\forall c : Contain \bullet$
$\quad \exists n1, n2 : (EntitySet \cup RelationshipSet) \bullet$
$\quad\quad (n1, n2) = contain(c)$

次に上で定義したERFモデルを用いてメタメタモデルを定義する．メタメタモデルにおいて，エンティティとして現われるのはENTITY, RELATIONSHIP, FIELDであり，これらを集合 *EntitySet* に属する変数として宣言する．そして，これらの要素を関数 *relateEtoE* によって関連付けている集合 *RelationshipSet* に属する変数が *relate*1, *relate*2, *contain*1, *contain*2, *contain*3 となる．

MetaMetaModel

$ERFModel$
$ENTITY, RELATIONSHIP, FIELD : NODE$
$relate1, relate2, contain1, contain2, contain3 : NODE$

$EntitySet = base(ENTITY)$
$RelationshipSet = base(RELATIONSHIP)$
$FieldSet = base(FIELD)$
$\{ENTITY, RELATIONSHIP, FIELD\}$
$\quad \subseteq EntitySet$
$\{relate1, relate2, contain1, contain2\}$
$\quad \subseteq RelationshipSet$
$(RELATIONSHIP, ENTITY) = relateEtoE(relate1)$
$(RELATIONSHIP, FIELD) = relateEtoE(relate2)$
$(FIELD, ENTITY) = relateEtoE(contain1)$
$(FIELD, RELATIONSHIP) = relateEtoE(contain2)$
$(FIELD, FIELD) = relateEtoE(contain3)$

最後に，メタメタモデルを使って記述するメタモデルにおける，各構成要素の生成/削除に関するスキーマを定義する．これらのスキーマも先程と同様に前節で示したメタモデル記述に対する指針に従っている．ノードタイプの構成要素となるエンティティとフィールドに対するスキーマを以下に示す．

CreateEntity

$\Delta ERFModel$
$entity? : NODE$

$entity? \notin EntitySet$
$EntitySet' = EntitySet \cup \{entity?\}$

DeleteEntity

$\Delta ERFModel$
$entity? : NODE$

$entity? \in EntitySet$
$EntitySet' = EntitySet \setminus \{entity?\}$

CreateField

$\Delta ERFModel$
$field? : NODE$

$field? \notin FieldSet$
$FieldSet' = FieldSet \cup \{field?\}$

DeleteField

$\Delta ERFModel$
$field? : NODE$

$field? \in FieldSet$
$FieldSet' = FieldSet \setminus \{field?\}$

アークタイプとして定義されるリレーションシップの生成/削除に対するスキーマは以下のようになる．

CreateRelationship

$\Delta ERFModel$
$relationship? : NODE$
$entity1? : NODE$
$entity2? : NODE$

$relationship? \notin RelationshipSet$
$entity1? \in EntitySet$
$entity2? \in EntitySet$
$relateEtoE' = relateEtoE \cup$
$\quad \{relationship? \mapsto (entity1?, entity2?)\}$

DeleteRelationship

$\Delta ERFModel$
$relationship? : NODE$

$relationship? \in RelationshipSet$
$relateEtoE' = \{relationship?\} \ndres relateEtoE$

3.5 図式モデルの仕様記述例

以上に示したメタモデルを記述するためのスキーマを使った，実際の図式モデルの形式的記述の例を挙げる．本報告では，例としてペトリネットモデルについて形式化する．まず，ペトリネットモデルのメタモデルを定義・形式化し，それを用いてペトリネットモデル（ベースモデル）を定義・形式化する．

ペトリネットメタモデル

メタメタモデルを使って表したペトリネットモデルのメタモデルを図??に示す．まずこの図の状態を形式化したスキーマ *PetriNetMetaModel* を以下のように定義する．

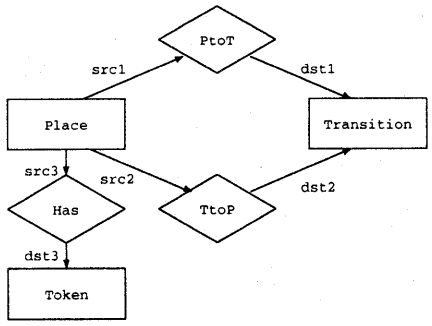

図 2: ペトリネットメタモデル

PetriNetMetaModel

$MetaMetaModel$
$Place, Transition, PtoT, TtoP, Token, Has : NODE$
$src1, src2, src3, dst1, dst2, dst3 : NODE$

$\{Place, Transition, PtoT, TtoP, Token\} \subseteq EntitySet$
$\{src1, src2, src3, dst1, dst2, dst3\} \subseteq RelationshipSet$
$(Place, PtoT) = relateEtoE(src1)$
$(PtoT, Transition) = relateEtoE(dst1)$
$(Transition, TtoP) = relateEtoE(src2)$
$(TtoP, Place) = relateEtoE(dst2)$
$(Place, Has) = relateEtoE(src3)$
$(Has, Token) = relateEtoE(dst3)$

次にペトリネットメタモデルによってペトリネットベースモデルを記述する際に必要な制約を記述する．これをスキーマ *PetriNetMetaModel*1 として定義する．ペトリネットモデルにおいて，ノードタイプの構成要素は *Place*, *Transition*, *Token* であり，アークタイプの構成要素は *PtoT*, *TtoP*, *Has* である．

*PetriNetMetaModel*1

$PetriNetMetaModel$
$PlaceSet, TransitionSet, PtoTSet,$
$\quad TtoPSet, TokenSet, HasSet : \mathbb{P}\, NODE$
$relatePlacetoTransition : NODE \nrightarrow (NODE \times NODE)$
$relateTransitiontoPlace : NODE \nrightarrow (NODE \times NODE)$
$relatePlacetoToken : NODE \nrightarrow (NODE \times NODE)$

$PlaceSet = base(Place)$
$TransitionSet = base(Transition)$
$PtoTSet = base(PtoT)$
$TtoPSet = base(TtoP)$
$TokenSet = base(Token)$
$HasSet = base(Has)$
$PtoTSet = \mathrm{dom}\, relatePlacetoTransition$
$(PlaceSet \times TransitionSet)$
$\quad = \mathrm{ran}\, relatePlacetoTransition$
$TtoPSet = \mathrm{dom}\, relateTransitiontoPlace;$
$(TransitionSet \times PlaceSet)$
$\quad = \mathrm{ran}\, relateTransitiontoPlace$
$HasSet = \mathrm{dom}\, relatePlacetoToken$
$(PlaceSet \times TokenSet)$
$\quad = \mathrm{ran}\, relatePlacetoToken$
$\forall ptot : PtoTSet;\ p : PlaceSet;\ t : TransitionSet \bullet$
$\quad relatePlacetoTransition(ptot) = (p, t) \Leftrightarrow$
$\quad\quad (\exists r : RelationshipSet \bullet relateEtoE(r)$
$\quad\quad\quad = (meta(p), meta(ptot))) \wedge$
$\quad\quad (\exists r : RelationshipSet \bullet relateEtoE(r)$
$\quad\quad\quad = (meta(ptot), meta(t)))$
$\forall ttop : TtoPSet;\ p : PlaceSet;\ t : TransitionSet \bullet$
$\quad relateTransitiontoPlace(ttop) = (t, p) \Leftrightarrow$
$\quad\quad (\exists r : RelationshipSet \bullet relateEtoE(r)$
$\quad\quad\quad = (meta(t), meta(ttop))) \wedge$
$\quad\quad (\exists r : RelationshipSet \bullet relateEtoE(r)$
$\quad\quad\quad = (meta(ttop), meta(p)))$
$\forall has : HasSet;\ p : PlaceSet;\ t : TokenSet \bullet$
$\quad relatePlacetoToken(has) = (p, t) \Leftrightarrow$
$\quad\quad (\exists r : RelationshipSet \bullet relateEtoE(r)$
$\quad\quad\quad = (meta(p), meta(has))) \wedge$
$\quad\quad (\exists r : RelationshipSet \bullet relateEtoE(r)$
$\quad\quad\quad = (meta(has), meta(t)))$

次にペトリネットモデルの各構成要素の生成/削除に関するスキーマを定義する．ペトリネットモデルにも指針で示したようにノードタイプとアークタイプの構成要素が存在するが，その中のプレースに対するスキーマ定義をノードタイプの例として挙げる．その他の構成要素に対しても名称が異なる以外は同じである．

CreatePlace

$\Delta PetriNetMetaModel1$
$place? : NODE$

$place? \notin PlaceSet$
$PlaceSet' = PlaceSet \cup \{place?\}$

DeletePlace

$\Delta PetriNetMetaModel1$
$place? : NODE$

$place? \in PlaceSet$
$PlaceSet' = PlaceSet \setminus \{place?\}$

アークタイプの例として，プレースからトランジションへのアークに対するスキーマを以下に示す．

CreatePtoT

$\Delta PetriNetMetaModel1$
$ptot? : NODE$
$place? : NODE$
$transition? : NODE$

$ptot? \notin PtoTSet$
$place? \in PlaceSet$
$transition? \in TransitionSet$
$relatePlacetoTransition' =$
$\quad relatePlacetoTransition$
$\quad \cup\{ptot? \mapsto (place?, transition?)\}$

DeletePtoT

$\Delta PetriNetMetaModel1$
$ptot? : NODE$

$ptot? \in PtoTSet$
$relatePlacetoTransition' = \{ptot?\}$
$\quad \lhd relatePlacetoTransition$

ペトリネットモデルは発火することによってモデルの状態を変化させることができる．発火という現象はプレース中に存在するトークンが別のトークンに移動することで表すことができる．つまり，モデル中のトークンという構成要素の削除と生成の記述を利用して発火の操作を形式化することができる．このことをふまえて発火を表すスキーマ *fire* を以下に示す．ここで，発火可能なトランジションは *transition?*，そのトランジションに対する入力プレースは *inplace*，出力プレースは *outplace* で表し，全ての *inplace* に対し関連しているトークンを1つ削除し，全ての *outplace* に対しトークンを1つ生成している．

fire

$\Delta PetriNetMetaModel1$
$transition? : NODE$
$outtokenset : \mathbb{P}\, NODE$

$transition? \in TransitionSet$
$outtokenset \cap TokenSet = \varnothing$
$\forall inplace : PlaceSet \mid$
$\quad \exists ptot : PtoTSet \bullet (inplace, transition?)$
$\quad\quad = relatePlacetoTransition(ptot) \bullet$
$\quad\quad \exists token : TokenSet \bullet$
$\quad\quad\quad \exists has : HasSet \bullet (inplace, token)$
$\quad\quad\quad\quad = relatePlacetoToken(has) \land$
$\quad\quad\quad TokenSet' = TokenSet \setminus \{token\} \land$
$\quad\quad\quad relatePlacetoToken' = \{has\}$
$\quad\quad\quad\quad \lhd relatePlacetoToken$
$\forall outplace : PlaceSet \mid$
$\quad \exists ttop : TtoPSet \bullet (transition?, outplace)$
$\quad\quad = relateTransitiontoPlace(ttop) \bullet$
$\quad\quad \exists outtoken : outtokenset \bullet$
$\quad\quad\quad \exists has : HasSet \bullet (outplace, outtoken)$
$\quad\quad\quad\quad = relatePlacetoToken(has) \land$
$\quad\quad\quad TokenSet' = TokenSet \cup \{outtoken\} \land$
$\quad\quad\quad relatePlacetoToken' = relatePlacetoToken$
$\quad\quad\quad\quad \cup\{has \mapsto (outplace, outtoken)\}$

ペトリネットベースモデル

図式モデルの形式化の最後にペトリネットのベースモデルを形式化する．ペトリネットのベースモデルは図 ?? にあるモデルを以下のように形式化した．ペトリネットという種類の図式モデルにおいて，他のペトリネットモデルを形式化する場合は，このベースモデルの部分のみを記述するだけで容易にペトリネットモデルの仕様を形式化することができる．

apetrinet

$PetriNetMetaModel1$
$place1, place2, place3 : NODE$
$trans1 : NODE$
$token1, token2 : NODE$
$arc1, arc2, arc3 : NODE$
$h1, h2 : NODE$

$\{place1, place2, place3\} \subseteq PlaceSet$
$\{trans1\} \subseteq TransitionSet$
$\{token1, token2\} \subseteq TokenSet$
$\{arc1, arc2\} \subseteq PtoTSet$
$\{arc3\} \subseteq TtoPSet$
$\{h1, h2\} \subseteq HasSet$
$(place1, trans1) = relatePlacetoTransition(arc1)$
$(place2, trans1) = relatePlacetoTransition(arc2)$
$(trans1, place3) = relateTransitiontoPlace(arc3)$
$(place1, token1) = relatePlacetoToken(h1)$
$(place1, token2) = relatePlacetoToken(h2)$

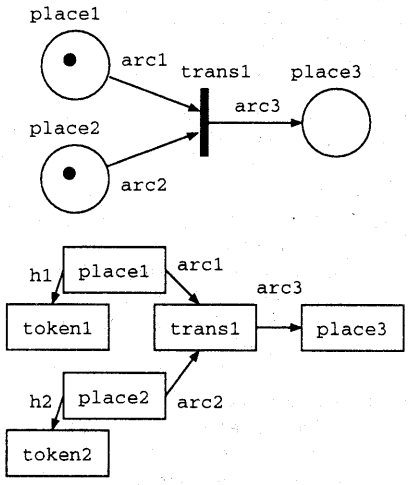

図 3: ペトリネットモデル

4 本アプローチによる検証方法について

前節で図式モデルに対する形式化のアプローチについて述べたが，これらを利用して，様々な検証が考えられる．

1つは，形式化されたモデルの状態がメタモデルの制約を満たしているかどうかの検証である．この制約を満たしているならば，それはその図式モデルにおいては正しいモデルであることが保証される．

そしてもう1つは，モデリングの作業が正しい手順で行われているかどうかの検証である．これは構成要素の生成/削除のスキーマを組み合わせることで調べることができる．例えば，存在しないノードへのアークによる関連づけや，その接続における型のチェックなどが調べられる．

最後に，種類の異なる図式モデル間の整合性に関する検証である．種類の異なる図式モデルでも，それぞれが表している意味が重複している場合がある．このような時，本アプローチでは，これらの図式モデルのメタモデルのスキーマと更に，それらのメタモデル間で満たされるべき整合性に関する制約を記述したスキーマを加えて調べれば容易に検証が行える．例えば，UML によって記述された仕様においてクラス図，状態遷移図，シーケンス図の間の整合性を検証するのに有効である．

5 おわりに

本研究では，ソフトウェア仕様における図式モデルに焦点を当て，これを形式化する手間を軽減するためにモデルをメタ階層化し記述のレベル分けをした．これによって，仕様記述者はメタモデルの制約の基でモデルを形式化するため，この作業が容易となる．また，メタモデルに関しては形式的記述に対して指針を与えたことで，ある程度手間を軽減することができる．

今後は，このアプローチによって様々な種類の図式モデルの形式化を試みると共に，このような検証機構を備えたツールの実現を考えている．

参考文献

[1] J. M. Spivery: The Z notation: A Reference Manual -Second Edition-, Programming Research Group University of Oxford, http://www.afm.sbu.ac.uk/z/ (1992)

[2] B. ポター等著, 田中武二 監訳: ソフトウェア仕様記述の先進技術—Z 言語, プレンティスホール・トッパン (1993)