JAIST Repository
https://dspace.jaist.ac.jp/

Title	オブジェクト指向方法論のための検証フレームワークに関する研究
Author(s)	花田，真樹
Citation	
Issue Date	1999-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1221
Rights	
Description	Supervisor:片山 卓也，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

オブジェクト指向方法論のための検証フレームワークに関する研究

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専攻

花田真樹

1999 年 2 月 15 日

Copyright © 1999 by Masaki Hanada

目 次

1 はじめに **1**

1.1 研究の背景と目的 1

1.2 論文の構成 2

2 形式的モデル **3**

2.1 基本モデル 4

2.1.1 基本オブジェクトモデル 4

2.1.2 基本動的モデル 7

2.1.3 基本機能モデル 10

2.2 統合モデル 11

2.3 例題 16

2.3.1 例題の仕様 16

2.3.2 例題の基本モデル 16

2.3.3 例題の統合写像 18

3 計算機による検証支援 **23**

3.1 HOL の概要 23

3.1.1 theory 構築のための基本的な ML 関数 25

3.1.2 基本的な導出規則 25

3.2 Well-formedness チェック 27

3.3 データの流れに注目した一貫性検証 29

3.3.1 概要 29

3.4 不変表明を用いた一貫性検証 33

3.4.1 概要 33

3.4.2 HOL を用いた検証支援 35

3.5 状態に注目した属性の値の性質に関する検証 47
3.5.1 HOL を用いた検証環境 49
3.5.2 属性の値に関する性質の証明 61

4 検証フレームワーク 68

4.1 検証フレームワーク 68
4.2 割り当て関数と翻訳関数 70
4.2.1 不変表明を用いた一貫性検証で構築された世界観 70
4.2.2 状態に注目した属性の値に関する検証で構築された世界観 72
4.3 検証フレームワークを用いた検証アプリケーションの作成 74
4.3.1 不変表明を用いた一貫性検証 76
4.3.2 状態に注目した属性の値に関する検証 76

5 おわりに 78

5.1 まとめ 78
5.2 今後の課題 78

図 目 次

2.1 形式化の方針 3
2.2 基本オブジェクトモデルの図的表現の例 5
2.3 ML 表記の例 (クラス) 5
2.4 ML 表記の例 (リンク属性) 5
2.5 ML 表記の例 (関連) 6
2.6 ML 表記の例 (集約関連) 6
2.7 ML 表記の例 (継承関係) 7
2.8 基本動的モデルの図的表現の例 8
2.9 ML 表記の例 (状態遷移) 9
2.10 基本機能モデルの図的表現の例 10
2.11 ML 記述 (データフロー) 11
2.12 所有者付き属性と所有者付き関数 12
2.13 限定子によるリンクの指定 13
2.14 基本モデルと統合モデル 15
2.15 例題の基本モデルの図的表現 16

3.1 継承関係 27
3.2 委譲の統合写像 28
3.3 動的モデルのおけるデータの流れと機能モデルの対応関係 29
3.4 公理 30
3.5 Transive 規則 31
3.6 Communication 規則 31
3.7 不変表明を用いた一貫性検証 33
3.8 不変表明を用いた一貫性検証の例題への適用 34
3.9 状態に注目した属性の値の性質に関する検証 47

3.10 オブジェクトシステム 48
3.11 状態遷移に対する公理 58
3.12 クラスからの実体化 61
3.13 初期状態 62
3.14 遷移 t0 の発火後の動作の状態と属性環境の状態 64
3.15 遷移 t1 発火後の動作状態と属性の状態 65
3.16 遷移 t1 発火後の動作状態と属性の状態 66

4.1 検証フレームワークを用いた検証アプリケーション 69
4.2 世界観の異なる割り当て関数、翻訳関数 69
4.3 検証アプリケーション作成のアーキテクチャ 75

第 1 章

はじめに

1.1 研究の背景と目的

現在、多くのオブジェクト指向方法論が提案されている。それらの方法論の１つにOMT法がある。OMT法では対象システムを記述するために３つのモデル (オブジェクトモデル、動的モデル、機能モデル) を用いる。これらの３つのモデルは、対象システムを直交する３つの視点に分割して記述する。しかし、ここで用いられるモデルは形式的取扱いが十分でないため、計算機による支援を困難にしている。

以上の問題点から、オブジェクト指向方法論のための**形式的モデル** [1](以下、形式的モデル) が提案されている。形式的モデルは、OMT 法をベースとして形式化を行ったものであり、集合と関数の概念を用いて定義されている。

本論文では、形式的モデルに基づいた検証を計算機により支援する手法を提案する。

検証を計算機上で支援するソフトウェアのことを検証アプリケーションと呼ぶ。形式的モデルに基づく検証として、データの流れに注目した一貫性検証、不変表明を用いた一貫性検証、状態に注目した属性の値に関する検証がある。データの流れに注目した一貫性検証は公理と推論規則による公理系より機能モデル記述されているデータフローを証明する形式で行われる。不変表明を用いた一貫性検証はクラスに対して高階述語論理を用いて不変表明を与え、アクションが実行されても不変表明を満たしているかどうか検証するものである。状態に注目した属性の値に関する検証は、ある状態における属性の値がどのような性質を持つかを高階述語論理を用いて検証するものである。

形式的モデルを用いた検証法は、これらの他にも無数にある。そこで、検証フレームワークという概念を用いて、検証アプリケーション作成のコストを削減し、有用な検証支援環境を構築することを目的とする。これによって、検証アプリケーションを作成する場合はホットスポットの部分を実装するだけで実現できる。

1.2 論文の構成

本論文では、1 章で論文の概要を説明する。2 章で本研究で用いる ML 上に実装した形式的モデルに関する概要を説明する。そして 3 章で計算機を用いてどのような検証支援が行えるかを説明する。4 章では検証フレームワークという概念を導入し、検証アプリケーション作成のアーキテクチャを示す。5 章で本研究のまとめを行う。

第 2 章

形式的モデル

この章では本研究に用いる形式的モデルについて説明する。形式的モデルは、OMT 法で用いられるシステムを 3 つの側面により記述する分析モデルを基礎として形式化を行なったものである。3 つのモデルを独立に定義した基本モデルと、独立に構築されたモデルの構成要素を**統合写像**の概念を用いて定義した統合モデルに分けられる。

- 基本モデル
  3 つのモデルをそれぞれ独立に定義
- 統合モデル
  独立に定義したモデルを統合写像の概念を用いて統合

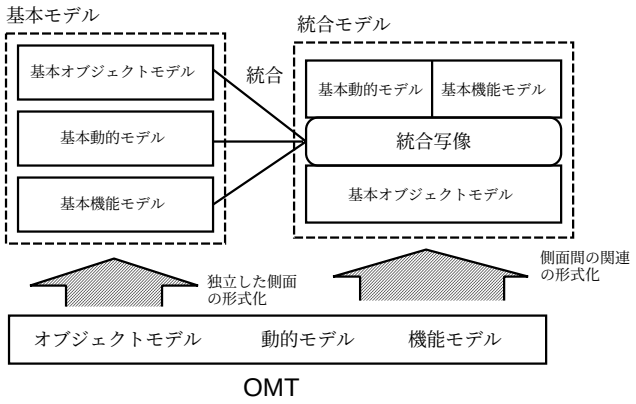


図 2.1: 形式化の方針

2.1 基本モデル

基本モデルは、OMT 法で用いられている 3 つのモデルを、各々のモデルで閉じた概念のみを用いて定義している。モデルの基本集合としてそれぞれの側面固有の概念を表現する識別子集合を用いる。識別子は各々の側面を構成する要素の最小単位であり、他の側面とは独立したものである。識別子は各々の側面に固有な概念を抽象化したものであるため、これらの持つ意味は別にドキュメイント化される。基本集合を用いてモデルを定義することにより、システムを３つの側面に独立に分解することができる。基本モデルは基本オブジェクトモデル、基本動的モデル、基本機能モデルにより構成される。検証支援環境では、これらの構築された形式的モデルは ML 上の参照型に格納される。

2.1.1 基本オブジェクトモデル

基本オブジェクトモデルでは、基本集合として、属性識別子の集合、関数識別子の集合、クラス識別子の集合、リンク識別子の集合、関連識別子の集合、集約識別子の集合、継承識別子の集合、限定識別子の集合が用いられる。

- クラス識別子の集合`ClassID`
- 関連識別子の集合`AssocID`
- リンク識別子の集合`LpropID`
- 集約識別子の集合`AggrID`
- 継承識別子の集合`InherID`
- 限定識別子`LimitID`
- 属性識別子の集合`AttrID`
- 関数識別子の集合`FuncID`

それぞれの識別子集合を文字列型で表現し、集合をリスト型で表現する。
以下のように文字列型を`ID`、リスト型を`Set` として定義する。

```
type ID = string
type 'a Set = 'a list
```

基本オブジェクトモデルは、対象システムに出現する属性、関数、クラス、関連、集約関係、継承関係により定義される。

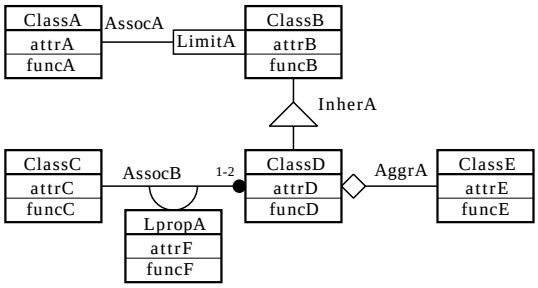
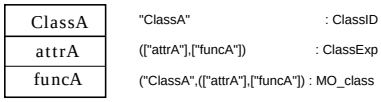


図 2.2: 基本オブジェクトモデルの図的表現の例

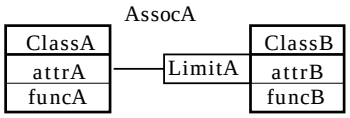
- クラスは、出現するクラスを識別するクラス識別子と、そのクラスがもつ属性識別子集合と関数識別子集合を示すクラス式`ClassExp` により表現される。この対応関係は写像`MO_class` によって表現される。

```
type ClassID = ID;
type ClassExp = AttrID Set * FuncID Set;
type MO_class = ( ClassID * ClassExp ) Set ;
```



- 関連は、出現する関連を識別する関連識別子と、その関連の内容を示す関連式`AssocExp`により表現される。この対応関係は写像`MO_assoc`によって表現される。ここで`"free"`とは限定子のないこと意味する限定識別子である。

```
type AssocID = ID;
type AssocExp = (ClassID * LimitID * MExp)
                * (ClassID * LimitID * MExp)
type MO_assoc = ( AssocID * AssocExp ) Set
```



"AssocA" : AssocID

("ClassA","free",(1,1)),
("ClassB","LimitA",(1,1)) : AssocExp

("AssocA",(("ClassA,"free",(1,1)),
("ClassB","limitA",(1,1))) : MO_assoc

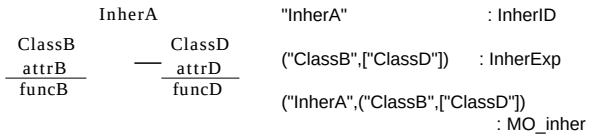


図 2.7: ML 表記の例 (継承関係)

基本オブジェクトモデル`ObjectModel` は、対象システムで用いられる各識別子集合と、その識別子が示す内容を表現する各々の写像によって定義される。

```
type ObjectModel = AttrID Set * FuncID Set * ClassID Set * LpropID Set
   * LimitID Set * AssocID Set * AggrID Set * InherID Set * MO;
```

以下、基本オブジェクトモデルの定義を示す。

```
type ClassExp = AttrID Set * FuncID Set;   (* 各々の式型の定義 *)
type LpropExp = AttrID Set * FuncID Set;
type MExp = int * int ;
type AssocExp = (ClassID * LimitID * MExp)
                * (ClassID * LimitID * MExp)
                * LpropID Set;
type AggrExp = ClassID * ( ClassID * MExp) list;
type InherExp = ClassID * ClassID list ;
type MO_class = ( ClassID * ClassExp ) Set ;(* 各々の写像型の定義 *)

type MO_lprop = ( LpropID * LpropExp ) Set ;
type MO_assoc = ( AssocID * AssocExp ) Set
type MO_aggr = ( AggrID * AggrExp ) Set ;
type MO_inher = ( InherID * InherExp ) Set ;
type MO = MO_class * MO_lprop * MO_assoc * MO_aggr * MO_inher;

(* 基本オブジェクトモデル型の定義 *)

type ObjectModel = AttrID Set * FuncID Set * ClassID Set * LpropID Set
   * LimitID Set * AssocID Set * AggrID Set * InherID Set * MO;
```

2.1.2 基本動的モデル

基本動的モデルでは、基本集合として、イベント識別子の集合、状態識別子の集合、アクション識別子の集合、遷移条件識別子の集合、状態遷移識別子の集合が用いられる。

- イベント識別子の集合`EventID`
- 状態識別子の集合`StateID`
- アクション識別子の集合`ActionID`

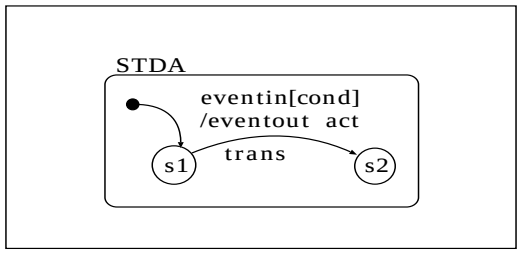


図 2.8: 基本動的モデルの図的表現の例

- 遷移条件識別子の集合CondID
- 状態遷移識別子の集合TransID
- 状態遷移図識別子STDID

文字列型をID、リスト型をSet として定義する。

```
type ID = string
type 'a Set = 'a list
```

動的モデルDynamicModel は、対象システムに出現する状態遷移図識別子集合とその識別子が示す内容を対応づける写像MD_dm によって表現される。

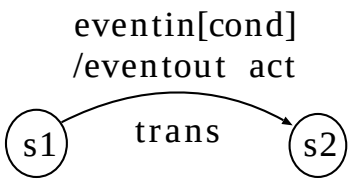
```
type DynamicModel = STDID Set * MD_dm ;
type MD_dm = ( STDID * ST ) Set;
```

状態遷移図は初期遷移状態と状態遷移の集まりであり、状態遷移図ST は対象システムに用いられるイベント識別子の集合、状態識別子の集合、アクション識別子の集合、遷移条件識別子の集合、状態遷移識別子の集合、初期状態と、状態遷移識別子との内容を表現する状態遷移式TransExp によって定義される。この対応関係は写像MD_trans によって表現される。

```
type ST  = StateID Set * EventID Set * EventID Set * ActionID Set
           * CondID Set * TransID Set  * MD_trans * StateID;
type MD_trans = (TransID * TransExp) Set;
```

遷移前状態"s1"、入力イベント式EExp_in "eventin"、遷移条件"cond"、出力イベント式["eexp_out"]、アクション"action"、遷移後状態"s2" により構成される場合、状態遷移は("s1",EExp_in"eventin","cond",["eexp_out"],"action","s2") の６つ組で定義される。これは、オブジェクトが状態"s1"である時、入力イベント式EExp_in "eventin"を充足するイベントを受取りかつ遷移条件"cond"が充足していると、イベント"eexp_out"を出力し、アクション"action"を実行し、状態"s2"に遷移することを意味する。

- 入力イベント式EExp_in はイベント識別子e1 を受け取り真になるEEXP_EVT "e1"、イベント識別子"e1"とイベント識別子"e2"を受け取り真になるEEXP_AND (EEXP_EVT"e1",EEXP_EVT"e2") 、イベン



・状態遷移前状態 "s1"　　: StateID

・入力イベント式
　EEXP_EVT "eventin"　　: EExp_in

・遷移条件識別子 "cond"　　: CondID

・出力イベント式
　["eventout"]　　: EventID list

・アクション識別子"act"　　: ActID

・状態遷移後状態 "s2"　　: StateID

・状態遷移式
　("s1",EEXP_EVT "eventin","cond",["eventout"],"s2")　:TransExp

　("trans",("s1",EEXP "eventin","cond",[eventout"],"s2") : MO_trans

図 2.9: ML 表記の例 (状態遷移)

ト識別子"e1"またはイベント識別子"e2"を受け取り真になるEEXP_OR (EEXP_EVT"e1",EEXP_EVT"e2")、無条件で真になるEEXP_EMPTY を表現している。

```
datatype EExp_in = EEXP_EVT of EventID (* 入力イベント式型はデータ型を
                 | EEXP_NOT of EventID      用いて定義する*)
                 | EEXP_AND of ( EExp_in * EExp_in)
                 | EEXP_OR of (EExp_in * EExp_in)
                 | EEXP_EMPTY;
```

- 出力イベント式はイベント識別子の集合で表現される。

```
type EExp_out = EventID list;
```

出力イベント式はイベント識別子の集合で表現される。これは、出力するイベント識別子の列を表現する。

基本動的モデルの定義は以下に示す。

```
type EventID = ID; (* 基本集合を文字列型として宣言する *)
type StateID = ID;
type ActionID = ID;
type CondID = ID;
type STDID = ID;
type TransID = ID;

datatype EExp_in = EEXP_EVT of EventID (* 入力イベント式型はデータ型を
                 | EEXP_NOT of EventID      用いて定義する*)
                 | EEXP_AND of ( EExp_in * EExp_in)
                 | EEXP_OR of (EExp_in * EExp_in)
                 | EEXP_EMPTY;
```

```
type EExp_out = EventID list;
type TransExp = (StateID * EExp_in * CondID * EExp_out
                                          * ActionID * StateID);

type MD_trans = (TransID * TransExp) Set;

type ST  = StateID Set * EventID Set * EventID Set * ActionID Set
 * CondID Set * TransID Set  * MD_trans * StateID;

type MD_dm = ( STDID * ST ) Set;

(* 基本動的モデル型の定義 *)
type DynamicModel = STDID Set * MD_dm ;
```

2.1.3 基本機能モデル

基本機能モデルでは、基本集合として、プロセス識別子の集合、アクター識別子の集合、データストア識別子の集合、データ識別子の集合、データフロー識別子の集合が用いられる。

- プロセス識別子の集合`ProcessID`
- アクター識別子の集合`ActorID`
- データストア識別子の集合`StoreID`
- データ識別子の集合`DataID`
- データフロー識別子の集合`FlowID`

文字列型を`ID`、リスト型を`Set`として定義する。

```
type ID = string
type 'a Set = 'a list
```

基本機能モデルは、対象システムにプロセス、アクター、データストア、データ、とデータフローによって定義される。

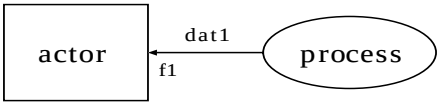
図 2.10: 基本機能モデルの図的表現の例

- データフローは、識別されるデータフロー識別子とその内容を表すデータフロー式`FlowExp`によって表現される。対応関係は写像`MF_flow`で示される。

```
type FlowID = ID;
type FlowExp = (Node * Node) * DataID;
type MF_flow = (FlowID * FlowExp) Set;
```

ノードはプロセス、アクター、データストアのどれかである。

```
datatype Node = PROCESS of ProcessID
              | ACTOR of ActorID
              | STORE of StoreID;
```



Actor "actor" : Node

Process "process" : Node

((Process "process", Actor "actor"),"data1") : FlowExp

("f1",((Process "process", Actor "actor"),"data1")) : MF_flow

図 2.11: ML 記述 (データフロー)

基本機能モデルに定義を以下に示す。

```
type ProcessID = ID; (* 基本集合を文字列型として宣言する *)
type ActorID = ID;
type StoreID = ID;
type DataID = ID;
type FlowID = ID;

datatype Node = PROCESS of ProcessID
              | ACTOR of ActorID
              | STORE of StoreID;

type FlowExp = (Node * Node) * DataID;

type MF_flow = (FlowID * FlowExp) Set;

(* 基本機能モデル型の定義 *)
type FunctionModel = ProcessID Set *  ActorID Set * StoreID Set *
                     DataID Set * FlowID Set * MF_flow;
```

2.2 統合モデル

独立に定義された基本モデルは、同じ対象システムを記述したものであるため、システムの同一の部分をモデル化しているものがある。そこで、意味が共通する部分を対応づける統合写像の概念を用いる。

統合写像は動的モデルに関するものと機能モデルに関するものがある。基本動的モデルに関する統合写像では、それぞれの動作の詳細が他の基本モデルとどのような関係にあるかを定義している。又、基本機能モデルに関する統合写像では、機能的側面からとらえた要素の詳細が他の基本モデルとどのような関係にあるかを定義している。

統合写像

前半は統合写像に表れる各々の式について説明する。

- AO は所有者付き属性であり、所有者はクラス識別子かリンク属性識別子かイベント識別子である。

```
datatype AO = AO_CLASS of ClassID * AttrID
            | AO_LPROP of LpropID * AttrID
            | AO_EVT   of EventID * AttrID ;
```

- FO は所有者付き関数であり、所有者はクラス識別子かリンク属性識別子である。

```
datatype FO = FO_CLASS of ClassID * FuncID
            | FO_LPROP of LpropID * FuncID ;
```

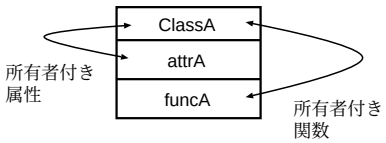


図 2.12: 所有者付き属性と所有者付き関数

- Term は所有者付き属性、又は所有者付き属性に対する所有者付き関数適用を示している。

```
datatype Term =  TERM_ATTR of AO
               | TERM_FUNC of (FO * Term list);
```

- BExp は遷移条件式であり、BEXP_TRUE は真、BEXP_FALSE は偽、BEXP_APP は関数適用、BEXP_NOT は引数の真なら偽を偽なら真を表現し、BEXP_AND は論理積、BEXP_OR は論理和、BEXP_LIMIT は限定子により限定されたリンクおよびオブジェクトが存在するかどうかを調べる条件を表現している。

```
datatype BExp =  BEXP_TRUE
               | BEXP_FALSE
               | BEXP_APP of (FO * Term list)
               | BEXP_NOT of (BExp)
               | BEXP_AND of (BExp * BExp)
               | BEXP_OR of (BExp * BExp)
               | BEXP_LIMIT of (FO * Term list * LimitID);
```

- AExp はアクション式であり、AEXP_APP は関数適用を、AEXP_SUBST は属性への代入を、AEXP_CONS はアクション式の連結を表現している。

```
datatype AExp = AEXP_APP of (FO * Term list)
              | AEXP_SUBST of (AO * Term)
              | AEXP_CONS of (AExp * AExp) ;
```

- LExp は限定式であり、LEXP_VAL は限定子が持つ値により特定されるリンクにイベントを出力することを意味し、LEXP_APP は限定子が持つ値を関数評価により特定する式であり、限定子が持つべき値を返す関数評価の返り値で特定されるリンクにイベントを出力することを表現している。SExp はイベントを出力するリンクを表現している。

```
datatype LExp = LEXP_VAL of (LimitID* string)
              | LEXP_APP of (FO * Term list * LimitID)
              | LEXP_OR of (LExp * LExp)
              | LEXP_AND of (LExp * LExp)
              | LEXP_NOT of LExp;
datatype SExp = SEXP_ASSOC of AssocID
```

以下の図のように限定子の値によってリンクの限定を行う。親機電話 1 は内線番号を指定することによって配送されるリンクが限定される。

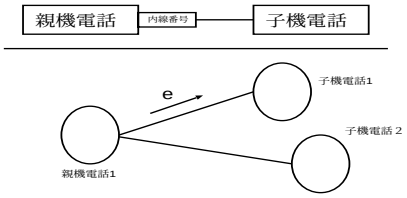


図 2.13: 限定子によるリンクの指定

- UST はクラスの振舞に関する統合写像であり、クラスとそのクラスに属しているオブジェクトの振舞が記述されている状態遷移図を対応づける写像である。

```
type UST = (ClassID * STDID) Set;
```

- Us は委譲の統合写像であり、委譲のタイミングを定義するための、部分オブジェクトが属するクラスと状態を対応付ける写像である。

```
type Us = ((STDID * StateID) * ClassID Set) Set;
```

- Uc は状態遷移の統合写像であり、条件と基本オブジェクトモデル中に出現する関数と属性を用いて定義したBExp を対応付ける写像である。

```
type Uc = (CondID * BExp) Set;
```

- Ua はアクションの統合写像であり、アクションと基本オブジェクトモデル中に出現する関数と属性を用いて定義したAExp を対応付ける写像である。

```
type Ua = (ActionID * AExp) Set;
```

- Ue はイベントへの属性付加の統合写像であり、イベントとそれに付加する属性集合を対応付ける写像である。

```
type Ue = (EventID * AttrID Set) Set;
```

- Ueo はイベント出力先の統合写像であり、イベント出力先を指定する写像である。

```
type Ueo = ((STDID * TransID * EventID) * SExp) Set;
```

- Ud はデータの統合写像であり、計算の対象であるデータと属性を対応付ける写像である。

```
type Ud = (DataID * AttrID) Set;
```

- Up はプロセスの統合写像であり、計算の変換を行うプロセスと関数を対応付ける写像である。

```
type Up = (ProcessID * FO) Set;
```

- Uac はアクターの統合写像であり、データの生産と消費を行うアクターと関数集合と属性集合を対応付ける写像である。

```
type Uac = (ActorID * ((AO Set) * (FO Set))) Set;
```

- Ust はデータストアの統合写像であり、データを格納するデータストアと関数集合と属性集合を対応付ける写像である。

```
type Ust = (StoreID * ((AO Set) * (FO Set))) Set;
```

以下に統合写像の定義を示す。

```
datatype AO = AO_CLASS of ClassID * AttrID
            | AO_LPROP of LpropID * AttrID
            | AO_EVT   of EventID * AttrID ;

datatype FO = FO_CLASS of ClassID * FuncID
            | FO_LPROP of LpropID * FuncID ;

datatype Term =  TERM_ATTR of AO
                | TERM_FUNC of (FO * Term list);

datatype BExp =  BEXP_TRUE
                | BEXP_FALSE
                | BEXP_APP of (FO * Term list)
                | BEXP_NOT of (BExp)
                | BEXP_AND of (BExp * BExp)
                | BEXP_OR of (BExp * BExp)
                | BEXP_LIMIT of (FO * Term list * LimitID);
datatype AExp = AEXP_APP of (FO * Term list)
                | AEXP_SUBST of (AO * Term)
                | AEXP_CONS of (AExp * AExp) ;
datatype LExp = LEXP_VAL of (LimitID* string)
                | LEXP_APP of (FO * Term list * LimitID)
                | LEXP_OR of (LExp * LExp)
```

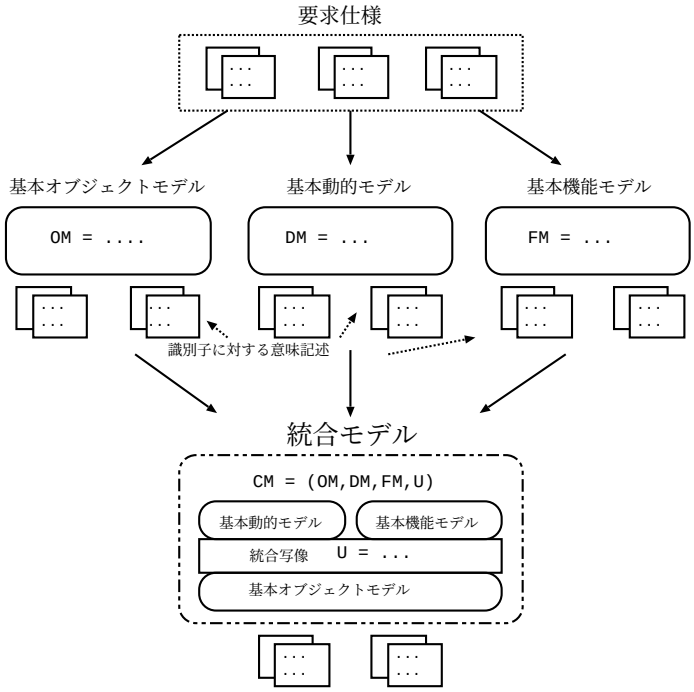


図 2.14: 基本モデルと統合モデル

```
                | LEXP_AND of (LExp * LExp)
                | LEXP_NOT of LExp;
datatype SExp = SEXP_ASSOC of AssocID
                | SEXP_LIMIT of (AssocID * LExp)
                | SEXP_CONS of (SExp * SExp);

type UST = (ClassID * STDID) Set;
type Us = ((STDID * StateID) * ClassID Set) Set;
type Uc = (CondID * BExp) Set;
type Ua = (ActionID * AExp) Set;
type Ue = (EventID * AttrID Set) Set;
type Ueo = ((STDID * TransID * EventID) * SExp) Set;
type Ud = (DataID * AttrID) Set;
type Up = (ProcessID * FO) Set;
type Uac = (ActorID * ((AO Set) * (FO Set))) Set;
type Ust = (StoreID * ((AO Set) * (FO Set))) Set;
type U = UST * Us * Uc * Ua * Ue * Ueo * Ud * Up * Uac * Ust; (* 統合モデル型
の定義 *)
type UM = (ObjectModel * DynamicModel * FunctionModel * U );
```

統合写像の概念を用いて、独立に分析された結果を対応づけることにより、モデル同士のすりあわせやトレードオフがおこなわれ、それぞれの側面を反映した 1 つの統合モデル`UM` を構成することができる。対応づけは、基本モデルとそれぞれの構成要素の意味記述を基におこなわれる。

2.3 例題

2.3.1 例題の仕様

対象システムは以下の仕様をもつ。

- 学生は論文を書いて出来上がると事務に提出する。
- 事務は期限内であれば受理し論文受理リストに論文を登録する。
- 事務は何人受理されているかを確認するために現在の受理人数も把握する。

分析モデルとして、形式的モデルの図による表記を以下に示す。

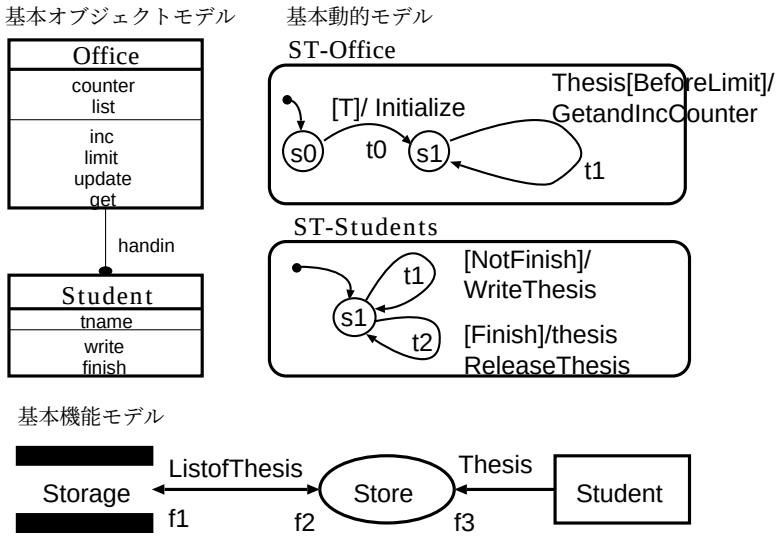


図 2.15: 例題の基本モデルの図的表現

2.3.2 例題の基本モデル

例題の基本オブジェクトモデルの最初の要素`["tname","counter","list"]` の型は`AttrID list` であり、例題で用いられる属性識別子の集合を表している。また、2番目の要素から8番目要素も、同様に例題で用いられる各々の集合を示している。9番目の要素は写像を示しており、各々の写像は組のリストで表現される。また、限定識別子`"free"`は限定しないことを示す識別子である。また、`MExp` はすべて自然数より大きい値を許すのでこれを負を用いて表現するものとする。

```
(* 例題の基本オブジェクトモデル *)
```

```
  (["tname","counter","list"],["write","finish","inc","limit","update","get"],
   ["Student","Office"],[],["free"],["handin"],[],[],
   ([("Student",(["tname"],["write","finish"])),
     ("Office",(["counter","list"],["inc","limit","update","get"]))],[],
    [("handin",(("Office","free",(1,1)),("Student","free",(0,~1)),[]))],[],[]))
  : ObjectModel
```

例題の基本動的モデルの最初の要素["ST_Student","ST_Office"] の要素の型はSTDID list であり、例題で用いられる状態遷移図識別子の集合である。"ST_Office"では初期状態が"s0"である。その初期状態から状態"s1"へは遷移"t0"で、状態"s1"から状態"s1"へは遷移"t1"で表現される。遷移"t0"は状態遷移式("s0",EEXP_EMPTY,"T",[],"Initialize","s1") に対応し、遷移"t1"は("s1",EEXP_EVT "thesis", "BeforeLimit",[],"GetandIncCounter","s1") に対応している。

```
(* 例題の基本動的モデル *)
  (["ST_Student","ST_Office"],
   [("ST_Student",
     (["s2"],[],["thesis"],["ReleaseThesis","WriteThesis"],
      ["Finish","NotFinish"],["t2","t3"],
      [("t2",("s2",EEXP_EMPTY,"NotFinish",["thesis"],"WriteThesis","s2")),
       ("t3",("s2",EEXP_EMPTY,"Finish",["thesis"],"ReleaseThesis","s2"))],
      "s2")),
    ("ST_Office",
     (["s0","s1"],["thesis"],[],["GetandIncCounter","Initialize"],
      ["BeforeLimit","T"],["t0","t1"],
      [("t0",("s0",EEXP_EMPTY,"T",[],"InitCounter","s1")),
       ("t1",
        ("s1",EEXP_EVT "thesis","BeforeLimit",[],"GetandIncCounter","s1"))],
      "s0"))]) : DynamicModel
```

例題には３つのフロー"f2","f1","f3"が存在する。これらはデータの流れ先を表し、"f2"はSTORE "Storage" からPROCESS "Store" に"ListofThesis"が流れることを意味している。また、"f1"はPROCESS "Store"からSTORE "Storage"にに"ListofThesis"が流れることを意味し、"f3"はActor "student"からPROCESS "Store" に"Thesis"が流れることを意味している。

```
(* 例題の基本機能モデル *)
```

```
(["Store"],["student"],["Storage"],["ListofThesis","Thesis"],
 ["f2","f1","f3"],
 [("f2",((STORE "Storage",PROCESS "Store"),"ListofThesis")),
  ("f1",((PROCESS "Store",STORE "Storage"),"ListofThesis")),
  ("f3",((ACTOR "student",PROCESS "Store"),"Thesis"))]) : FunctionModel
```

2.3.3 例題の統合写像

統合モデルでは、独立に定義されたそれぞれの基本モデルを統合写像という概念を用いて統合を行う。例題に表れる統合写像について説明する。

- 統合写像UST

  ```
  [("Student","ST_Student"),("Office","ST_Office")]
  ```

  ("Student","ST_Student")、("Office","ST_Office") は、クラス"Student"はそのクラスに属しているオブジェクトの振舞が記述されている状態遷移図"ST_Student"と対応づけられ、クラス"Office"はそのクラスに属しているオブジェクトの振舞が記述されている状態遷移図"ST_Office"と対応づけられる。

- 統合写像Uc

  ```
  [("T",BEXP_TRUE),
      ("NotFinish",BEXP_NOT (BEXP_APP (FO_CLASS ("Student","finish"),[]))),
      ("Finish",BEXP_APP (FO_CLASS ("Student","finish"),[])),
      ("BeforeLimit",BEXP_APP (FO_CLASS ("Office","limit"),[]))]
  ```

  条件と基本オブジェクトモデル中に出現する関数と属性を用いて定義した条件遷移論理式BExp を対応付ける写像である。T はtrue と、NotFinish はStudent.finish() と、Finish はStudent.finish() と、BeforeLimit はOffice.limit() と対応づけられる。[1]。

- 統合写像Ua

  - ("Initialize",
    AEXP_CONS
    (AEXP_SUBST

[1] 以下では一般的な記法で記す

```
          (AO_CLASS ("Office","counter"),
           TERM_FUNC (FO_CLASS ("Office","0"),[])),
        AEXP_SUBST
          (AO_CLASS ("Office","list"),
           TERM_FUNC (FO_CLASS ("Office","Nil"),[]))))
```

"Initialize"は意味記述より「カウンタを初期化し、論文リストを初期化する」を意味しており、基本オブジェクトモデルの識別子"counter"と"0"、"list"と"Nil"を用いて定義したアクション式AExp と対応づけられる。アクション式はOffice.counter := 0 ;Office.list := []である。

```
–     ("WriteThesis",
       AEXP_SUBST
         (AO_EVT ("thesis","tname"),
          TERM_FUNC (FO_CLASS ("Student","write"),[])))
```

"WriteThesis"は意味記述より「論文を書くアクション」を意味している。

Student.tname := Student.write()

と対応づけられる。

```
–     ("GetandIncCounter",
       AEXP_CONS
         (AEXP_CONS
            (AEXP_SUBST
               (AO_CLASS ("Office","counter"),
                TERM_FUNC
                  (FO_CLASS ("Office","inc"),
                   [TERM_ATTR (AO_CLASS ("Office","counter"))])),
             AEXP_APP
               (FO_CLASS ("Office","get"),
                [TERM_ATTR (AO_EVT ("thesis","tname"))])),
        AEXP_SUBST
          (AO_CLASS ("Office","list"),
           TERM_FUNC
             (FO_CLASS ("Office","update"),
              [TERM_ATTR (AO_CLASS ("Office","list")),
```

```
                    TERM_ATTR (AO_EVT ("thesis","tname"))]))))]
```

"GetandIncCounter"は、意味記述より「論文を受理してを counter をインクリメントする」アクションを表現しており、

```
Office.counter := Office.inc(Office.counter);Office.get(thesis.tname);
Office.list := Office.update(Office.list,thesis.tname)
```

と対応づけられる。

- 統合写像Ue

```
[("thesis",["tname"])]
```

イベント"thesis"へ属性"tname"が付加していること表現する。

- 統合写像Ueo

```
[(("ST_Student","t1","thesis"),SEXP_ASSOC "handin")]
```

("ST_Student","t1","thesis") のイベント出力先が関連識別子"handin"であることを表現する。

- 統合写像Ud

```
[("ListofThesis","list"),("Thesis","tname")]
```

データ"ListofThesis"が属性"list"、データ"Thesis"が属性"tname"に対応づけられる。

- 統合写像Up

```
[("Store",FO_CLASS ("Office","update"))],
```

プロセス"Store"が所有者付き関数FO_CLASS ("Office","update") に対応づけられる。

- 統合写像Uac

```
[("student",
  ([AO_CLASS ("Student","tname")],
   [FO_CLASS ("Student","write"),FO_CLASS ("Student","finish")]))],
```

アクター"student"が([AO_CLASS ("Student","tname")], [FO_CLASS ("Student","write"), FO_CLASS ("Student","finish")]) に対応づけられる。

- 統合写像Ust

```
[("Storage",
  ([AO_CLASS ("Office","counter"),AO_CLASS ("Office","list")],[]))]
```

データストア"Storage"が([AO_CLASS ("Office","counter"),AO_CLASS ("Office","list")],[])に対応づけられる。

Office	事務を表現するクラス
Student	学生を表現するクラス
counter	カウンタの値を保持する属性
list	受理した論文のタイトルのリストを格納する属性
inc	counter をインクリメントする関数
limit	期限内だと真を、期限を過ぎるを偽を返す関数
get	論文を受け取る関数
update	論文のタイトルリストと論文のタイトルを引数とし、 その論文を追加した論文のタイトルリストを返す関数
tname	論文のタイトルを保持する属性
finish	論文を書き終えると真になる関数
write	論文を書き、タイトルを返す関数
handin	学生が事務に論文を渡す関連
thesis	論文の受渡しを示すイベント
GetandIncCounter	論文を受理してを counter をインクリメントするアクション
Initialize	カウンタとリストの初期化を行うアクション
WriteThesis	論文を書くアクション
ReleaseThesis	論文をリリースするアクション
BeforeLimit	期限以内かを調べる条件
Finish	論文を書き終わったかを調べる条件
NotFinish	論文を書き終わっていないかを調べる条件
Storate	論文リストを保持しるデータストア
Store	論文を論文リストにストアするプロセス
Student	学生を表現するアクター
Thesis	論文を表現するデータ

第 3 章

計算機による検証支援

形式的モデル上で適用可能なものとしてすでに提案されているデータの流れに注目した一貫性検証、不変表明を用いた一貫性検証、状態に注目した一貫性検証に対し、どのような検証支援が行えるかの説明をする。検証を行う環境として ML と定理証明系 HOL を用いる。始めに HOL の概要を示す。次に各々の検証法の概要を説明し、例題に対して検証を行う。

3.1 HOL の概要

HOL は、理論のモジュール性、ユーザー定義型の作成が可能、ML 環境側からの柔軟な扱いが可能といった特徴をもっている。以下に HOL の概要を示す。

- 高階論理上で証明を行うシステム。
  高階述語論理をサポートしており、変数として関数や述語を用いることで柔軟な論理式の定義を可能にしている。

- forward proof と goal-directed proof をサポートしている。
  forward proof だけでなく、goal-directed proof もサポートしており、より戦略的な証明を行える。

- 関数型言語 ML 上で実装されている。
  HOL は関数型言語 ML 上で実装されており、ML に関する機能はすべて用いることが可能である。また、HOL 上での操作も ML の関数として用意されており、それらは、項や論理式の定義を行う場合、証明を行う場合、システムを扱う場合など多くに及んでいる。

- 組み込みの理論やライブラリが豊富である。
  arithmetic、list、set 等の理論やライブラリが多く、それらのすべてを利用可能である。また、一般

的な論理とは違う部分として、自ら証明を行って証明したものも、theory 上に定理として導入でき、後に利用可能である。

HOL 論理の項は、ML 上でterm と呼ばれる抽象型により表現される。これらは、(--‘ ‘--) の間に入力される。HOL 論理の項の型は、hol_type と呼ばれる ML の型を形成している。これらは、(--‘: ...‘--) で表現される。型は builtin 関数のtype_of で確認できる。これは、ML の型term ->hol_type をもち、項の論理上の型を返す。

HOL の項の論理上の型をオブジェクト言語の型、ML の式の ML 上の型をメタ言語の型としている。例えば、(--‘(1,T)‘--) はメタ言語の型term をもつ ML の式であり、オブジェクト言語の型(--‘:num # bool‘--) 型をもつ項と評価される。項は、変数、定数、関数適用、ラムダ計算の 4 種類がある。

以下に HOL の表記を示す。

Terms of HOL Logic			
King of term	*HOL notation*	*Standard notation*	*Description*
Truth	T	$\top$	*true*
Falsity	F	$\bot$	*false*
Negation	t̃	$\neg$t	*not t*
Disjunction	t_1\/t_2	t_1 *veet*$_2$	t_1 *or* t_2
Conjunction	t_1/\t_2	t_1 *wedget*$_2$	t_1 *and* t_2
Implication	t_1==>t_2	$t_1 \Rightarrow t_2$	t_1 *implies* t_2
Equality	t_1=t_2	$t_1 = t_2$	t_1 *equals* t_2
$\forall$-quantification	!x.t	$\forall x.t$	*for all x :t*
$\exists$-quantification	?x.t	$\exists x.t$	*for some x:t*
ε-term	@x.t	$\varepsilon x.t$	*an x such t hat :t*
Conditional	(t=>t_1|t_2)	$(t \rightarrow t_1, t_2)$	*if t then* t_1 *else* t_2

theory は、type、constants、definition、axiom を含んでいる。HOL 論理は definition、axiom から証明される theorem のリストを含んでいる。

以下に theory を構築するための基本的な ML 関数と導出規則を示す。

3.1.1 theory 構築のための基本的な ML 関数

```
new_theory : string -> unit
```

- new_theory "thy"を実行すると、thy という名前の新しい theory が作られる。現在いた theory は thy の新しい親の theory となる。

```
new_parent : string -> unit
```

- new_parent "thy"を実行すると、現在の theory の親の theory としてthy が加えられる。

```
new_open_axiom : string * term -> thm
```

- new_parent ("name",t) を実行すると、名前name で現在の theory に公理として加えられる。

```
new_definition : string * term -> thm
```

- new_definition ("name",(--`f t_1 ,...,t_n = t`--)) を実行すると、名前name で現在の theory に definition として加えられる。また、f はイコールをみたす新しい定数と宣言される。

- theory の definition は、一貫性を保証している axiom の特別な集まりであり、definition の共通した形は、f t_1,..., t_n = t である。f は項定数であり、t は項である。

3.1.2 基本的な導出規則

```
UNDISCH : thm -> thm
```

$$\frac{\Gamma \vdash t}{\Gamma, t' \vdash t}$$

```
SYM : thm -> thm
```

$$\frac{\Gamma \vdash t_1 \Rightarrow t_2}{\Gamma, t_1 \vdash t_2}$$

```
EQT_ELIM : thm -> thm
```

$$\frac{\Gamma \vdash t = T}{\Gamma \vdash t}$$

```
EQT_INTRO : thm -> thm
```

$$\frac{\Gamma \vdash t}{\Gamma \vdash t = T}$$

```
SPEC : term -> thm -> thm
```

$$\frac{\Gamma \vdash \forall x.t}{\Gamma \vdash t[t'/x]}$$

```
EXISTS : (term # term) -> thm -> thm
```

$$\frac{\Gamma \vdash t_1[t_2]}{\Gamma \vdash \exists t_1[x]}$$

```
CONJ : thm -> thm -> thm
```

$$\frac{\Gamma_1 \vdash t_1 \Gamma_2 \vdash t_2}{\Gamma_1 \cup \Gamma_2 \vdash t_1 \wedge t_2}$$

```
CONJUNCT1 : thm -> thm

CONJUNCT2 : thm -> thm
```

$$\frac{\Gamma \vdash t_1 \wedge t_2}{\Gamma \vdash t_1 \Gamma \vdash t_2}$$

```
CONJ1 : term -> thm -> thm
```

$$\frac{\Gamma \vdash t_1}{\Gamma \vdash t_1 \vee t_2}$$

```
CONJ2 : term -> thm -> thm
```

$$\frac{\Gamma \vdash t_2}{\Gamma \vdash t_1 \vee t_2}$$

3.2 Well-formedness チェック

Well-formedness のチェック事項を挙げる。

- 継承関係にループが存在しないことを示す制約。

  継承関係`InherExp` は`("ClassA",[("ClassB",(1,1))])` のように表現される。
  `inher1 = ("ClassA",[("ClassB",(1,1))])`、`inher2 = ("ClassB",[("ClassA",(1,1))])` とおくと、`#1 inher1 = #1 (hd (#2 inher2))` が真になるのように関係が存在してはいけないことになる。

  1. 全ての継承関係 `[inher1,inher2,...,inhern]` を集合 R とおく。集合 R' に集合 R を代入する。
  2. 集合 R' の先頭の要素`inher1` と２番目の要素`inher2` であらたな親子関係が存在する場合、`#2 inher1` に`#2 inher2` を加える。この処理を集合の全通りについて行う。これを集合 T に代入する。
  3. 全通り完了した時点で、集合 T にループ関係すなわち`#1 inher` が`#2 inher` に含まれていないかチェックする。含まれている時は偽を返す。含まれていない時は集合 R' と集合 T を比較し、同じ場合は真を返す。同じでない場合は集合 R' に集合 T を代入し、2 に戻る。

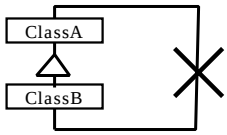


図 3.1: 継承関係

- 委譲の統合写像`Us` に関する制約。

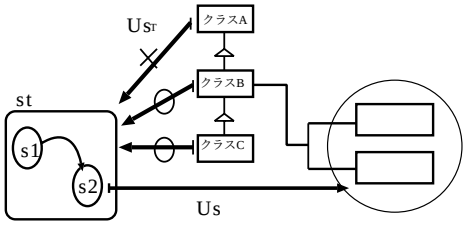


図 3.2: 委譲の統合写像

統合写像Us を((std,state),class1) とおくと、class1 からの集約関係先class2 とした時、その class2 の親であるクラス集合の中にstd から写像Ust 関係にあるクラスが含まれてはいてはいけない。これをすべての統合写像Us に対してチェックする。

- アクション式に関する制約。

  アクション式AExp において、「属性はその属性特有の値を保持する」という仮定を崩すような、代入の左辺と右辺に同じ属性識別子をもつ式を定義してはいけない。したがって、属性代入式において、左右に同じ文字列をもつ形式があった時に偽を返す。

- イベント出力先の統合写像に関する制約。

  統合写像Ueo を(st,t,e) とおくと、e は必ず状態遷移図st の出力イベント集合に含まれていなければならない。これがすべての統合写像Ueo に対してチェックする。

以上のように、形式的モデルの各々の制約に対してはそれをチェックする手順が存在する。

3.3 データの流れに注目した一貫性検証

3.3.1 概要

データの流れに注目した一貫性検証は、公理と推論規則から構成される公理系から、機能モデルに記述されているデータフローの証明をする形式で行われる。公理は基本動的モデルに記述されているデータの流れを基に、推論規則はモデルのセマンティクスから導かれるデータの流れを基に作成される。

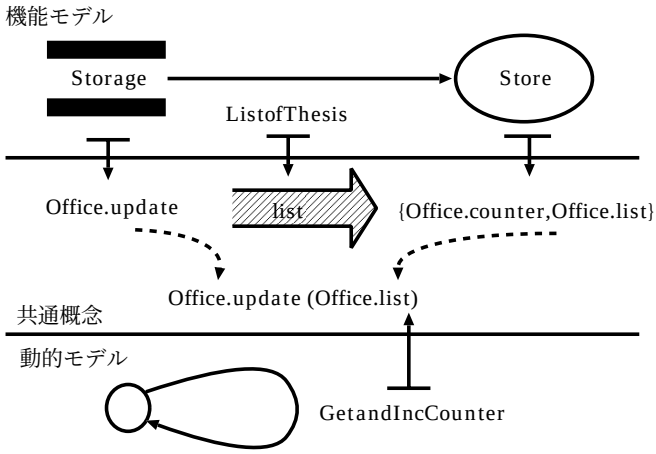


図 3.3: 動的モデルのおけるデータの流れと機能モデルの対応関係

公理は基本動的モデルに記述されているデータの流れを、規則はモデルのセマンティクスから導出されるデータの流れを表現するものである。この公理系で基本機能モデルに記述されているすべてのデータフローが証明可能であるとき、モデルのデータの流れの意味において一貫しているものとする。

公理

公理は基本動的モデルに記述されているデータの流れを表現するものであり、属性計算によりデータの流れが場合をすべて抽出することにより定義される。属性計算が記述されているのはアクション識別子に対するアクション式AExp、状態条件識別子に対する条件式BExp、イベント出力先式SExp である。

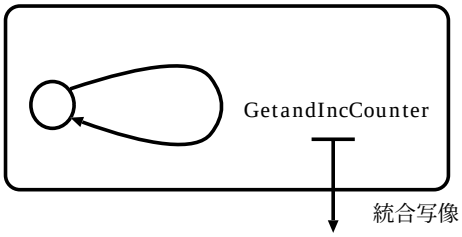
Pre からPost へ属性a が流れる属性フロー式AFExp は以下のとおりである。

$$\mathtt{AFExp} \stackrel{\text{def}}{=} \mathtt{Pre} \xrightarrow{\mathtt{d}} \mathtt{Post}$$

$$\mathtt{pre} \stackrel{\text{def}}{=} \mathtt{c.a}|\mathtt{c.f}|\mathtt{c.e.a}$$

$$\mathtt{post} \stackrel{\text{def}}{=} \mathtt{c.a}|\mathtt{c.f}|\mathtt{c.e.a}$$

ここで、e はイベント識別子、c はクラス識別子又はリンク識別子、f は関数識別子、a,d は属性識別子である。c.e.a はクラスc で用いられるイベントe に付属する属性a を表現する。



Office.counter := Office.inc(Office.counter);....

・関数 Office.inc は counter の値を参照
・属性 Office.counter は Office.inc の値を受ける

公理

$\vdash$ Office.counter $\xrightarrow{\text{counter}}$ Office.inc

$\vdash$ Office.inc $\xrightarrow{\text{counter}}$ Office.counter

図 3.4: 公理

図 3.4 では、関数が属性値を参照する場合と属性に関数の返り値を代入する場合の 2 つの公理に分けられる。

例題のクラス"Office"には以下のAExp がある。

```
Office.counter := 0 ;Office.list := []
Office.counter := Office.inc(Office.counter);Office.get(thesis.tname);
Office.list := Office.update(Office.list,thesis.tname)
```

これらから、以下の公理が作成される。

$$\vdash \texttt{Office.0} \xrightarrow{\texttt{counter}} \texttt{Office.counter}$$

$$\vdash \texttt{Office.[]} \xrightarrow{\texttt{counter}} \texttt{Office.list}$$

$$\vdash \texttt{Office.counter} \xrightarrow{\texttt{counter}} \texttt{Office.inc}$$

$$\vdash \texttt{Office.inc} \xrightarrow{\texttt{counter}} \texttt{Office.counter}$$

$$\vdash \texttt{Student.thesis.tname} \xrightarrow{\texttt{tname}} \texttt{Office.get}$$

$$\vdash \texttt{Office.update} \xrightarrow{\texttt{tname}} \texttt{Office.list}$$

$$\vdash \texttt{Office.liststackreltname} \rightarrow \texttt{Office.update}$$

$$\vdash \texttt{Student.thesis.tname} \xrightarrow{\texttt{tname}} \texttt{Office.update}$$

推論規則

以上で定義した公理は、属性や関数の参照/代入関係による直接的な属性の流れを定義したが、属性を介した推移的な属性の流れやイベントを介した推移的な属性値の流れを考慮していない。属性を介した推移的な属性値の流れは Transive 規則により定義され、イベントを介した推移的な属性値の流れは Communication 規則により表現される。

Transive 規則

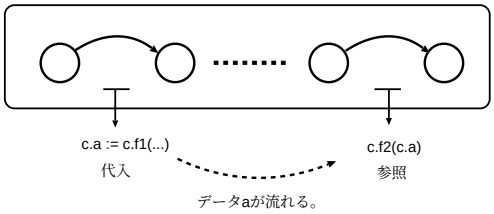


図 3.5: Transive 規則

Transive 規則は同一のデータ同士でデータが推移的に流れることを表現する。これは、属性に代入された値はオブジェクトが存在する間保持され、他のデータに変換されない間はそのデータが流れるとみなすことである。

Transive 規則は以下に定義される。

$$\frac{\mathtt{pre} \xrightarrow{\mathtt{d}} \mathtt{c.a}] \quad \mathtt{c.a} \rightarrow \mathtt{post}}{\mathtt{pre} \xrightarrow{\mathtt{d}} \mathtt{post}}$$

Communication 規則

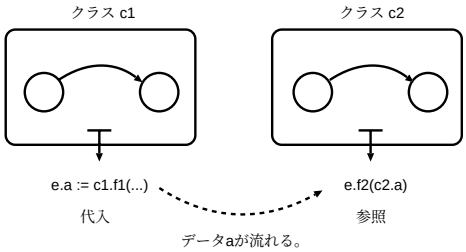


図 3.6: Communication 規則

Communication 規則はイベントを介してデータが送られる場合を表現するものである。イベントはリンクを介して出力するため、オブジェクト間でイベント通信が行われるためには、それらの間にリンクが存在しなければならない。そこでクラス $\mathtt{c_1}$ と $\mathtt{c_2}$ にリンクが張られている条件を $[\mathtt{LC}(\mathtt{c_1}, \mathtt{c_2})]$ とする。

$$\frac{[\mathtt{LC}(\mathtt{c}_1,\mathtt{c}_2)] \quad \mathtt{pre} \xrightarrow{\mathtt{d}} \mathtt{c}_1.\mathtt{e.a} \quad \mathtt{c}_2.\mathtt{e.a} \xrightarrow{\mathtt{d}} \mathtt{post}}{\mathtt{pre} \xrightarrow{\mathtt{d}} \mathtt{post}}$$

一貫性

データの流れ $\mathtt{pre} \xrightarrow{\mathtt{d}} \mathtt{post}$ が証明可能の場合、$\vdash \mathtt{pre} \xrightarrow{\mathtt{d}} \mathtt{post}$ とする。

基本機能モデルのデータフロー式が((Pre,Post),Dat)が証明可能であることをProvable((Pre,Post),Dat)と定義すると以下が成立する時である。

$$\mathtt{Provable}((\mathtt{Pre},\mathtt{Post}),\mathtt{Dat}) \leftrightarrow \exists \mathtt{pre} \in \mathtt{U}(\mathtt{Pre}).\exists \mathtt{U}(\mathtt{Post}). \vdash \mathtt{pre} \rightarrow \mathtt{Post}$$

これらには以下のアルゴリズムを用いて自動証明が可能である。

属性フロー式 $\mathtt{Pre} \xrightarrow{\mathtt{d}} \mathtt{Post}$ が証明可能かどうかを調べるためにPreから公理、Transive規則、Communication規則を適用し、到達可能な集合を作り、Postがその中に含まれておけばよい。

1. 動的モデルから公理の集合Aを作成する。
2. 属性フロー式より $\mathtt{Pre} \xrightarrow{\mathtt{d}}$ であるものを抽出し、集合Rとする。
3. 集合R'、集合Tを空集合にしておく。
4. Rの中の各々の要素を選ぶ。その要素から、$R \cup A$ の各々の要素に対しTransive規則、Communication規則を適用し得られたものをR'に加える。
5. R'に含まれていてRに含まれないものをTに加える。
6. RにTを加える。
7. Rの要素に $\mathtt{Pre} \xrightarrow{\mathtt{d}}$ が含まれるとプログラムをとめ、真をかえす。
8. Tが空になると3に戻る。

3.4 不変表明を用いた一貫性検証

3.4.1 概要

不変表明を用いた一貫性検証は、クラスの静的な性質を表現する不変表明が、基本動的モデルで定義される動作を実行しても成立しているかを検証するものである。

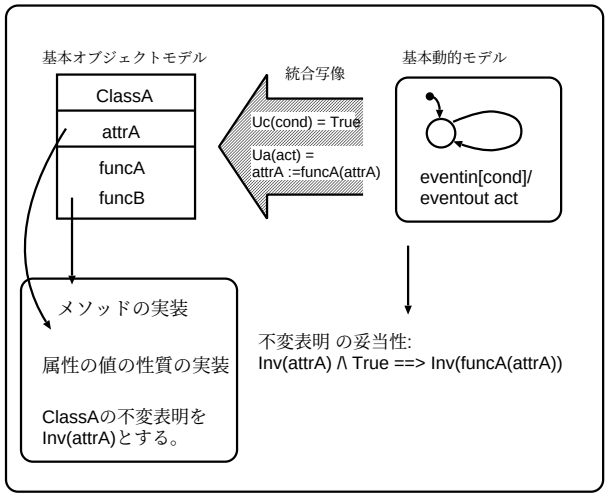


図 3.7: 不変表明を用いた一貫性検証

統合写像が与えられた動的モデルでは、アクションが実行されると、属性の値が変更される場合がある。基本オブジェクトモデルで定義した不変表明は、この属性の値を基に構成されているため、アクションが実行されて値が変更されても不変表明を満たしているがどうかを証明して、その妥当性を保証しなければならない。

クラスに対する不変表明を`Inv`、クラスを`c` とすると不変表明の妥当性は以下を証明すれば良い。
遷移`t` に対して、

- 遷移状態前が初期状態の場合

$$\mathtt{T} \Rightarrow \mathtt{Inv(c)}$$

- 遷移`t` が初期状態からの遷移でない場合

$$\mathtt{Inv(c)} \wedge \mathtt{bexp} \Rightarrow \mathtt{Inv(c)}$$

ここでは、条件遷移論理式を`bexp` とする。

次に、論文受理システムに対して検証を行う。クラス"Office"に対して不変表明として「"counter"の値は 0 以上であり、"list"に保持されている要素の数は"counter"と同じである」を割り当てる。

クラス"Office"に対応する状態遷移図の遷移"t0"が発火したとき、アクション"Initialize"の実行によって属性が書換えられても不変表明を満たしているか、また、遷移"t1"が発火したとき、アクション"GetandIncCounter" の実行によって属性が書換えられても不変表明を満たしているかを検証する。

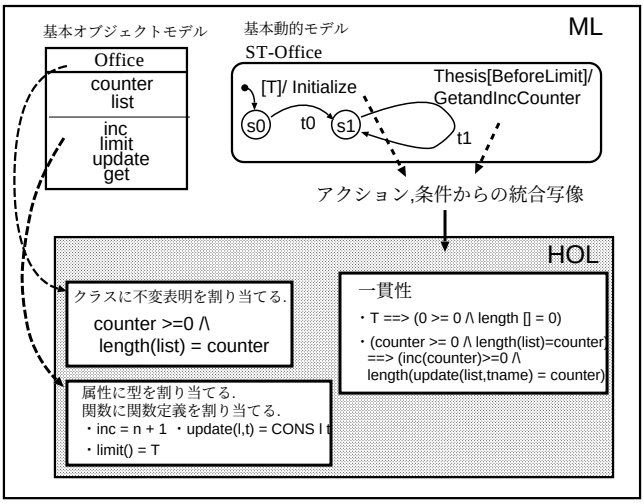


図 3.8: 不変表明を用いた一貫性検証の例題への適用

この検証法では識別子の意味記述を基に、属性識別子に型をもつ項定数を、関数識別子に対して HOL を用いた関数定義を割り当てる。また、クラス識別子に対しても不変表明を割り当てる。

- 属性識別子に対する割り当て

  基本オブジェクトモデルに出現する属性識別子に対して以下を割り当てる。

  - AO_CLASS ("Office","counter") に対して、(==‘:num‘==) 型の(--‘counter‘--) を割り当てる。
  - AO_CLASS ("Office","list") に対して、(==‘:string list ‘==) 型の(--‘list‘--) を割り当てる。
  - AO_CLASS ("Student","tname") に対して、(==‘:string‘==) 型の(--‘tname‘--) を割り当てる。

- 関数識別子に対する割り当て

  基本オブジェクトモデルに出現する関数識別子に対して以下を割り当てる。

- FO_CLASS ("Office","inc") に対して(--`inc(n:num)=n+1`--) を割り当てる。
- FO_CLASS ("Office","limit") に対して(--`limit(day:num)=day < pre_defined_limit`--) を割り当てる。
- FO_CLASS ("Office","update") に対して(--`update(ls:string list,s)=s::ls`--) を割り当てる。
- FO_CLASS ("Office","get") に対して(--`get(s:string)=(current_thesis = s)`--) を割り当てる。

- クラス識別子に対して割り当て

  クラス識別子に関して、以上で定義したものを用いて HOL を用いた不変表明を割り当てる。不変表明は、意味的に「counter は 0 以上であり、list に保持されている要素の数は counter と同じである」ことを示している。

  - "Office" に対して (--`0 <= counter /\ length (list) = counter`--)

統合写像が与えられた動的モデルではアクションが実行されると属性の値が更新される場合がある。基本オブジェクトモデルで定義した不変表明は、この属性の値を基に構成されているため、アクションが実行されて値が変更されても不変表明を満たしているかを証明し、妥当性を保証しなければならない。

- 不変表明の妥当性

  不変表明の妥当性とは全ての状態遷移に対して不変表明を仮定して動作を行っても、その表明が成立しているということである。クラス"Office"には"t0"と"t1"が存在するのでそれぞれに対して不変表明の妥当性を証明する。

  以下に、モデルの一貫性が成立するための 2 つの不変表明が定義される。

  - 状態遷移t0

    (--`T ==> 0 <= 0 /\ (LENGTH [] = 0)`--) : term

  - 状態遷移t1

    ```
    (--`0 <= counter /\ (LENGTH list = counter) /\ limit ==>
      0 <= inc counter /\ (LENGTH (update (list,tname)) = inc counter)`--)
    ```

3.4.2 HOL を用いた検証支援

HOL を用いて不変表明を用いた一貫性検証を支援するためには、構築したモデルを、それらの意味に沿って HOL 上に翻訳して、そのモデルと等価な意味をもつ HOL 上の世界を構築しなければならない。まず、

モデルの属性識別子や関数識別子に割り当てたもの概念を HOL 上に宣言する。
新しい理論の名前をinv として、定義する。

```
- new_theory "inv";
Declaring theory "inv".
val it = () : unit
```

親のtheory として、すでに HOL の組み込みのtheory string を宣言する。これにより、theory string を用いる事が可能となる。theory HOL は、14 個の先祖をもっており、それらの中に自然数のtheory であるarithmetic、リストのtheory｜である\verblist—が含まれている。

```
- new_parent "string";
val it = () : unit
```

属性識別子AO_CLASS ("Office","counter") に対して割り当てられた(==`:num`==) 型を持つ項定数(--`counter`--) を宣言する。AO_CLASS ("Office","list") に対して割り当てられた(==`:string list`==) 型を持つ項定数(--`list`--) を、AO_CLASS ("Student","tname") に対して割り当てられた(==`:string`==) 型を持つ項定数(--`tname`--) を宣言する。

```
- new_constant {Name = "counter", Ty = (==`:num`==)};
val it = () : unit
- new_constant {Name = "list", Ty = (==`:string list`==)};
val it = () : unit
- new_constant {Name = "tname", Ty = (==`:string`==)};
val it = () : unit
```

FO_CLASS ("Office","inc") とFO_CLASS ("Office","update") に対して割り当てられた(--`inc(n:num)=n+1`--) と(--`update ((ls:string list),(l:string)) = CONS l ls`--) を以下に宣言する。これらは、関数型であり、new_definition を用いて宣言する。

```
- new_definition ("inc", (--`inc n:num = n + 1`--));
val it = |- !n. inc n = n + 1 : thm
- new_definition
 ("update", (--`update ((ls:string list),(l:string)) = CONS l ls`--));
val it = |- !ls l. update (ls,l) = CONS l ls : thm
```

FO_CLASS ("Office","limit") とFO_CLASS ("Office","get") に割り当てられた関数定義を宣言したいのだが、"limit"と"get"は外部の環境を参照、更新する関数であり、前もって外部の環境を作成する。外部の環境の詳細は別に予め用意することにする。これにより、その環境を参照する定義を行うことが可能になる。この例では、外部の環境を表現する項定数(--`Epre_defined_limit`--)、(--`Eday`--)、(--`Ecurrentpaper`--) を導入する。(--`Epre_defined_limit`--) に対しては予め〆切の日を日数で表現するものとして自然数の型をもつ項定数として宣言する。(--`Eday`--) は、今日の日付を表すものとして自然数型をもつ項定数として宣言する。(--`Ecurrentpaper`--) は、論文のタイトルを表し、文字列型ををもつ項定数として宣言する。

```
- new_constant {Name = "Epre_defined_limit", Ty = (==`:num`==)};
val it = () : unit
- new_constant {Name = "Eday", Ty = (==`:num`==)};
val it = () : unit
- new_constant {Name = "Ecurrentpaper", Ty = (==`:string`==)};
val it = () : unit
```

次に、(--`Epre_defined_limit`--)、(--`Eday`--)、(--`Ecurrentpaper`--) は予め与えておく。(--`Epre_defined_limit`--) は締切日を意味し、予めに(--`10`--) として宣言する。(--`Eday`--) は今日の日付を意味し、(--`5`--) と宣言する。

```
- new_open_axiom ("Epre_defined_limit", (--`Epre_defined_limit = 10`--));
val it = |- Epre_defined_limit = 10 : thm
- new_open_axiom ("Eday", (--`Eday = 5`--));
val it = |- Eday = 5 : thm
```

外部の環境を整えたので以下の定義が可能になる。

FO_CLASS ("Office","limit") とFO_CLASS ("Office","get") にに対して割り当てられた関数定義を以下に宣言する。

```
- new_definition ("limit",  (--`limit = Eday <= Epre_defined_limit`--));
val it = |- limit = Eday <= Epre_defined_limit : thm
- new_definition ("get", (--`get t:string = t`--));
val it = |- !t. get t = t : thm
```

(--`get`--) は恒等関数として定義し、(--`Ecurrentpaper`--) に反映されるものとする。構築されたモデルから HOL 上の理論に翻訳するときは、以下の作業が必要となる。

- 外部の環境は、予め定義を行っておく。
- HOL 上で定義されていない型等は実装しておく必要がある。

例題のモデルに関して必要な要素を定義したtheory inv は以下である。

```
- print_theory "inv";
Theory: inv
Parents:
    string
    HOL
Type constants:
Term constants:
    Epre_defined_limit (Prefix)   :num
    Eday (Prefix)   :num
    Ecurrentpaper (Prefix)   :string
    limit (Prefix)   :bool
    get (Prefix)   :string -> string
    update (Prefix)   :string list # string -> string list
    inc (Prefix)   :num -> num
    counter (Prefix)   :num
    list (Prefix)   :string list
    tname (Prefix)   :string
Axioms:
    Epre_defined_limit |- Epre_defined_limit = 10
    Eday |- Eday = 5
Definitions:
    limit |- limit = Eday <= Epre_defined_limit
    get |- !t. get t = t
    update |- !ls l. update (ls,l) = CONS l ls
    inc |- !n. inc n = n + 1
Theorems:
```

クラス"Office"の不変表明についての証明を行う。各々の不変表明を、状態遷移"t1"に対してprop_t0、状態遷移t2 に対してprop_t1 とする。

```
- val prop_t0 = (--`T ==> 0 <= 0 /\ (LENGTH([]:string list) = 0)`--);
val prop_t0 = ``T ==> 0 <= 0 /\ (LENGTH [] = 0)`` : term

- val prop_t1 = (--`0 <= counter /\ (LENGTH list = counter) /\ limit ==>
    0 <= inc counter /\ (LENGTH (update (list,tname)) = inc counter)`--);
val prop_t1 =
``0 <= counter /\ (LENGTH list = counter) /\ limit ==>
  0 <= inc counter /\ (LENGTH (update (list,tname)) = inc counter)``:term
```

ここで、LENGTH はtheory list で定義されている定理である。関数add_theory_to_sml を用いて、適当な ML 上の名前に割り当てられる。ここでは、LENGTH という名前に割り当てられており、その定義は以下である。

```
- LENGTH;
val it = |- (LENGTH [] = 0)
/\ (!h t. LENGTH (CONS h t) = SUC (LENGTH t)) : thm
- type_of (--`LENGTH`--);
val it = ``:'a list -> num`` : hol_type
```

以下で、prop_t0 の証明とprop_t1 の証明を行う。証明手法として forward Proof と Goal Directerd Proof を用いて証明を行う。forward Proof とは公理と推論規則を用いて証明を行う手法である。Goal Directerd Proof については後で説明する。

prop_t0 の証明とprop_t1 の証明は次に示す。

prop_t0 の証明

HOL は、 forward proof と Goal directed proof をサポートしており、forward proof を行うための 3 つのプリミティブな推論規則として、ASSUME、DISCH、MP がある。これらを用いて導出規則を作成でき、forward proof では最もよく用いられる ML 関数である。

ASSUME の型はterm -> thm であり、term 型の t をとり $t \vdash t$ を推論する関数である。。

DISCH はの型はterm -> thm -> thm であり、term 型の t_1をとり$\cdots t_1 \cdots \vdash t_2$から$\cdots \vdash t_1 \Rightarrow t_2$を推論する関数である。

MP は以下である。

$$\frac{\Gamma_1 \vdash t_1 \Rightarrow t_2 \qquad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t}$$

その他に重要な導出規則として、REWRITE_RULE がある。

REWRITE_RULE は

$$\Gamma \vdash (u_1 = v_1) = \cdots = (u_n = v_n)$$

と

$$\Delta \vdash t$$

をとり、t における u_iを対応する v_iに変換する規則である。これは適用できなくなるまで行われる。これには、theory arithmetic のMULT_CLAUSES、ADD_CLAUSES が含まれている。その他にトートロジーチェッカーもある。

prop_t0 の証明に関しては forward proof を用いた証明を行う。

```
- prop_t0;
val it = ``T ==> 0 <= 0 /\ (LENGTH [] = 0)`` : term
```

定理LENGTH にCONJUNCT1 を適用し、th1 とする。次に、0 <= 0 にARITH_CONV とEQT_ELIM を適用し、th2 とする。ARITH_CONV は、arithmetic library の中で定義されており、自然数に関する項がT と等価であるかの証明を試みる。term が証明可能な場合、 term = T を返す。

```
- val th1 = CONJUNCT1 LENGTH;
val th1 = |- LENGTH [] = 0 : thm

- val th2 = EQT_ELIM(ARITH_CONV (--`0 <= 0`--));
val th2 = |- 0 <= 0 : thm
```

上の２つの定理に対してCONJ を適用し、th3 とする。

```
- val th3 = CONJ th2 th1;
val th3 = |- 0 <= 0 /\ (LENGTH [] = 0) : thm
```

T に対してASSUME を適用し、th4 とする。th3 とth4 にCONJ とCONJUNCT2 を適用する。

```
- val th4 = ASSUME (--`T`--);
val th4 = [T] |- T : thm
```

```
- val th5 = CONJUNCT2(CONJ th4 th3);
val th5 = [T] |- 0 <= 0 /\ (LENGTH [] = 0) : thm
```

th5 にDISCH を適用する。

```
- val theorem_t0 = DISCH (--`T`--) th5;
val theorem_t0 = |- T ==> 0 <= 0 /\ (LENGTH [] = 0) : thm
```

以上より、prop_t0 は証明される。

prop_t1 の証明

prop_t1 に関しても forward proof を用いて証明を行う。以下にインタラクティブに証明を行ったものを示す。prop_t0 と同様に行える。

```
val prop_t1 =
``0 <= counter /\ (LENGTH list = counter) /\ limit ==>
  0 <= inc counter /\ (LENGTH (update (list,tname)) = inc counter)``:term
```

LESS_EQ_ADD に対し、SPEC を用いて(--`counter`--)、(--`1`--) に置き換える。

```
- val th1 = (SPEC (--`1`--) (SPEC (--`counter`--) LESS_EQ_ADD));
val th1 = |- counter <= counter + 1 : thm
```

LESS_EQ_ADD の定理は以下である。これはtheory arithmetic の定理である。

```
- LESS_EQ_ADD;
val it = |- !m n. m <= m + n : thm
```

次に、LESS_EQ_ADD に対し、SPEC を用いて(--`0`--)、(--`counter`--)、(--`counter + 1`--) の順に置き換える。

```
- val th2 = (SPEC (--`counter + 1`--) (SPEC (--`counter`--)
                                 (SPEC (--`0`--) LESS_EQ_TRANS)));
val th2 = |- 0 <= counter /\ counter <= counter + 1 ==> 0 <= counter + 1
: thm
```

LESS_EQ_ADD の定理は以下である。これはtheory arithmetic の定理である。

```
- LESS_EQ_TRANS;
val it = |- !m n p. m <= n /\ n <= p ==> m <= p : thm
```

ASSUME (--`0 <= counter`--) を行って得られた定理に対して、CONJ を用いて、th1 と論理和をとる。MP を用い、 DISCH を用いて、前提を仮定に持ってくる。

```
- val th3 = DISCH (--`0 <= counter`--)
               (MP th2 (CONJ (ASSUME (--`0 <= counter`--)) th1));
val th3 = |- 0 <= counter ==> 0 <= counter + 1 : thm
```

REWRITE_RULE は１番目の引数である定理のリストを用いてth3 を書換える。ここで、th4 は最後に命題を導く時に用いる。

```
- val th4 = REWRITE_RULE [SYM (SPEC (--`counter`--) inc)] th3;
val th4 = |- 0 <= counter ==> 0 <= inc counter : thm
```

EQ_MONO_ADD_EQ に対し、SPEC を用いて(--`1`--)、(--`counter`--)、(--`LENGTH list`--) の順に置き換える。

```
- val th5 = ASSUME (--`(LENGTH list = counter)`--);
val th5 = [LENGTH list = counter] |- LENGTH list = counter : thm


- val th6 = SYM (SPECL [(--`LENGTH list`--),(--`counter`--),
                                    (--`1`--)] EQ_MONO_ADD_EQ);
val th6 = |- (LENGTH list = counter) = LENGTH list + 1
                                          = counter + 1 : thm
```

LESS_EQ_ADD の定理は以下である。これはtheory arithmetic の定理である。

```
- EQ_MONO_ADD_EQ;
val it = |- !m n p. (m + p = n + p) = m = n : thm
```

REWRITE_RULE [th6] を用いて、th5 を書換える。

```
- val th7 = REWRITE_RULE [th6] th5;
val th7 = [LENGTH list = counter] |- LENGTH list + 1 =
                                          counter + 1 : thm
```

次にADD1 に対して、SPEC を用いて(--`LENGTH list`--) に置き換える。SYM を用いて等式の左右を入れ替える。

```
- val th8 = SYM (SPEC (--`LENGTH list`--) ADD1);
val th8 = |- LENGTH list + 1 = SUC (LENGTH list) : thm
```

REWRITE_RULE [th8] を用いてth7 を書換える。

```
- val th9 = REWRITE_RULE [th8] th7;
val th9 = [LENGTH list = counter] |-
                            SUC (LENGTH list) = counter + 1 : thm
```

PURE_REWRITE_RULE は１番目の引数の定理のみを用いて次の引数を書換える。SPECL とは引数の順番に置き換える関数である。

```
- val th10 = PURE_REWRITE_RULE [SYM (REWRITE_CONV [LENGTH]
(--`LENGTH (CONS tname list)`--))] th9;
val th10 = [LENGTH list = counter] |-
 LENGTH (CONS tname list) = counter + 1 : thm


- val th11 = REWRITE_RULE
            [(SYM ((SPECL [(--`list`--),(--`tname`--)] update))),
             (SYM (SPEC (--`counter`--) inc))] th10;
val th11 =
  [LENGTH list = counter] |- LENGTH (update (list,tname)) =
                                                 inc counter : thm
```

DISCH を用いて前提条件を仮定に持ってくる。

```
- val th12 = DISCH (--`LENGTH list = counter`--) th11;
val th12 =
  |- (LENGTH list = counter) ==>
(LENGTH (update (list,tname)) = inc counter) : thm
```

ここからは、MP を用いて導いていく。

```
- val th13 = MP th4 (CONJUNCT1
(ASSUME (--`0 <= counter /\ (LENGTH list = counter) /\ limit`--)))
val th13 =
  [0 <= counter /\ (LENGTH list = counter) /\ limit] |-
                                            0 <= inc counter : thm
- val th14 = MP th12 ((CONJUNCT1 o CONJUNCT2)
(ASSUME (--`0 <= counter /\ (LENGTH list = counter) /\ limit`--)))
```

```
val th14 =
  [0 <= counter /\ (LENGTH list = counter) /\ limit]
  |- LENGTH (update (list,tname)) = inc counter : thm
- val theorem_t1 =
  DISCH (--`0 <= counter /\ (LENGTH list = counter) /\ limit`--)
  (CONJ th14 th15);

val theorem_t1 =
  |- 0 <= counter /\ (LENGTH list = counter) /\ limit ==>
     0 <= inc counter /\ (LENGTH (update (list,tname)) =
                                                 inc counter) : thm
```

以上で、クラス"Office"の不変表明が成立することを示すためにprop_t0とprop_t1をHOL上で証明した。次に、 forward Proof より効率的に行う証明法として Goal Directerd Proof を用いた証明を行う。Goal Directerd Proof は tactic と呼ばれる概念を用いて、証明を行う。tactics は以下の２つのことを行う関数である。goal を subgoal に分ける。また、 goal が subgoal に分けられる reason をつくる。これを justification と呼ぶ。Goal Directed Proof では、証明すべき命題を goal に設定し、tactic を適用することにより subgoal に分解する。そして、すべての subgoal が正しいと判明した時、justification をたどることにより、ゴールに設定された命題が正しいことを得る。

$A \wedge B$を証明するためには、A と Bを証明すれば良いことになる。

$$\frac{A \wedge B}{A \qquad B}$$

この tactic の justification は、$\vdash A$ かつ$\vdash B$の時、$\wedge$-introduction 規則

$$\frac{\Gamma_1 \vdash t_1 \qquad \Gamma_2 \vdash t_2}{\Gamma_1 \cup \Gamma_2 \vdash t_1 \wedge t_2}$$

を用いて、$\vdash A \wedge B$を得ることができる。

Goal Directerd Proof は命題をset_goal で与え、tactic を用いて証明を行う。

```
- set_goal([],prop_t1);
val it =
  Status: 1 proof.
  1. Incomplete:
       Initial goal:
       ``0 <= counter /\ (LENGTH list = counter) /\ limit ==>
```

```
        0 <= inc counter /\ (LENGTH (update (list,tname)) = inc counter)``
   : proofs

- e(REWRITE_TAC [Eday,Epre_defined_limit,limit]);
OK..
1 subgoal:
val it =
  ``0 <= counter /\ (LENGTH list = counter) /\ 5 <= 10 ==>
    0 <= inc counter /\ (LENGTH (update (list,tname)) = inc counter)``
   : goalstack
(* ...... *)
- e(REWRITE_TAC [SYM (SPEC (--`counter`--) ADD1)]);
OK..
1 subgoal:
val it =
  ``0 <= counter ==> 0 <= SUC counter``
  ----------------------------
      ``LENGTH list = counter``
   : goalstack
- e(REWRITE_TAC [DISCH_ALL
(MP (SPECL [(--`0`--),(--`SUC counter`--)] LESS_IMP_LESS_OR_EQ)
    (UNDISCH (SPECL [(--`0`--),(--`counter`--)] LESS_EQ_IMP_LESS_SUC)))]);
OK..

Goal proved.
|- 0 <= counter ==> 0 <= SUC counter
Goal proved.
|- 0 <= counter ==> 0 <= counter + 1
Goal proved.
[0 <= counter] |- 0 <= counter + 1
Goal proved.
[LENGTH list = counter, 0 <= counter]
|- 0 <= counter + 1 /\ (LENGTH list + 1 = counter + 1)
```

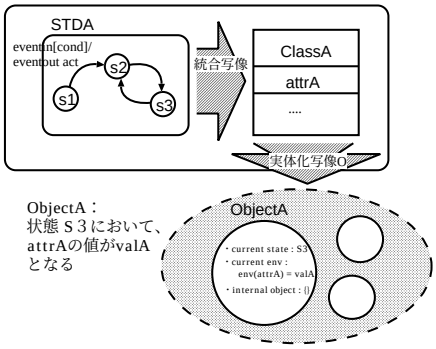
```
(* .....*)

Goal proved.
|- 0 <= counter /\ (LENGTH list = counter) ==>
   0 <= counter + 1 /\ (LENGTH (CONS tname list) = counter + 1)
Goal proved.
|- 0 <= counter /\ (LENGTH list = counter) /\ 5 <= 10 ==>
   0 <= counter + 1 /\ (LENGTH (CONS tname list) = counter + 1)
Goal proved.
|- 0 <= counter /\ (LENGTH list = counter) /\ 5 <= 10 ==>
   0 <= inc counter /\ (LENGTH (update (list,tname)) = inc counter)
val it =
  Initial goal proved.
  |- 0 <= counter /\ (LENGTH list = counter) /\ limit ==>
     0 <= inc counter /\ (LENGTH (update (list,tname)) = inc counter)
  : goalstack
```

3.5 状態に注目した属性の値の性質に関する検証

状態に注目した属性の値の性質に関する検証とは、ある状態における属性の値がどのような性質を持つかを高階述語論理を用いて検証するものある．オブジェクトの振舞いを公理を用いて表現する。

各々の状態遷移に対する事前条件、事後条件を用いた公理を作成する。調べたい性質を命題として HOL 上で与え、それに対して証明を行うことになる。



- AO_CLASS ("Office","counter") に対して、(==‘:num‘==) 型の(--‘counter‘--) を割り当てる。
- AO_CLASS ("Office","list") に対して、(==‘:string list ‘==) 型の(--‘list‘--) を割り当てる。
- AO_CLASS ("Student","tname") に対して、(==‘:string‘==) 型の(--‘tname‘--) を割り当てる。

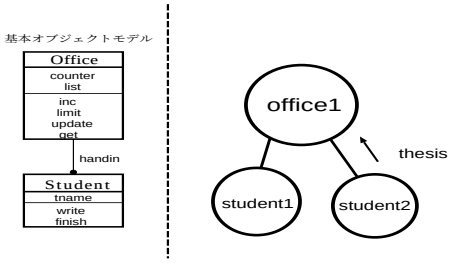
- 関数識別子に対する割り当て

  基本オブジェクトモデルに出現する関数識別子に対して以下を割り当てる。

  - FO_CLASS ("Office","inc") に対して(--‘inc(n:num)=n+1‘--) を割り当てる。
  - FO_CLASS ("Office","limit") に対して(--‘limit(day:num)=day < pre_defined_limit‘--) を割り当てる。
  - FO_CLASS ("Office","update") に対して(--‘update(ls:string list,s)=s::ls‘--) を割り当てる。
  - FO_CLASS ("Office","get") に対して(--‘get(s:string)=(current_thesis = s)‘--) を割り当てる。

- 基本動的モデル、オブジェクトシステムの基本的な概念に対しては、基本集合等のモデルと等価である HOL 上での実装が必要となる。

  - オブジェクトシステムは形式的モデルから実体化される、オブジェクト、リンク、メッセージなどにより構成される。



この検証法においては、オブジェクト識別子集合、状態識別子集合、イベント識別子集合、イベントインスタンス識別子集合を型として導入する。また、オブジェクトの属性の状態、動作の状態を以下の関数によって表現する。

- クラス"Office"から実体化されるオブジェクトの属性環境を表現する関数を定義する。"counter"に対してはOffice_env_conter、"list"に対してはOffice_env_list を用いる。これらは引数がオブジェクト、返り値がその属性に対する値である。
- オブジェクトの状態を表現する関数Office_state を導入する。これは引数がオブジェクト、返り値がその動作の状態である。
- オブジェクトが受け取るイベントインスタンス集合を表現する関数Office_input を導入する。これは引数がオブジェクト、返り値がそのオブジェクトが受け取るイベントインスタンス集合である。

これらは以下の章で詳細に説明する。

3.5.1 HOL を用いた検証環境

不変表明を用いた一貫性検証の場合と同様に、以下の順番で宣言、定義を行う。ここで、HOL では対象変数は、新しい変数として宣言せずにその変数を用いる時に変数として明示的に示すことになる。

```
- new_theory "env";
Declaring theory "env".
val it = () : unit
- new_parent "string";
val it = () : unit
```

- 新しい theory ”env”の宣言をする。
- 親の theory として、すでに HOL の組み込みの theory string を宣言する。これにより、theory string を用いる事が可能となる。

FO_CLASS ("Office","inc") とFO_CLASS ("Office","update") に対して割り当てられた(--‘inc(n:num)=n+1‘--) と(--‘update ((ls:string list),(l:string)) = CONS l ls‘--) を以下に宣言する。これらは、関数型であり、new_definition を用いて宣言する。

```
- new_definition ("inc", (--‘inc n:num = n + 1‘--));
val it = |- !n. inc n = n + 1 : thm
- new_definition
```

```
 ("update", (--`update ((ls:string list),(l:string)) = CONS l ls`--));
val it = |- !ls l. update (ls,l) = CONS l ls : thm
```

FO_CLASS ("Office","limit") とFO_CLASS ("Office","get") に割り当てられた関数定義を宣言したいのだが、"limit"と"get"は外部の環境を参照、更新する関数であり、前もって外部の環境を作成する。外部の環境の詳細は別に予め用意することにする。これにより、その環境を参照する定義を行うことが可能になる。この例では、外部の環境を表現する項定数(--`Epre_defined_limit`--)、(--`Eday`--)、(--`Ecurrentpaper`--) を導入する。(--`Epre_defined_limit`--) に対しては予め〆切の日を日数で表現するものとして自然数の型をもつ項定数として宣言する。(--`Eday`--) は、今日の日付を表すものとして自然数型をもつ項定数として宣言する。(--`Ecurrentpaper`--) は、論文のタイトルを表し、文字列型ををもつ項定数として宣言する。

```
- new_constant {Name = "Epre_defined_limit", Ty = (==`:num`==)};
val it = () : unit
- new_constant {Name = "Eday", Ty = (==`:num`==)};
val it = () : unit
- new_constant {Name = "Ecurrentpaper", Ty = (==`:string`==)};
val it = () : unit
```

次に、(--`Epre_defined_limit`--)、(--`Eday`--)、(--`Ecurrentpaper`--) は予め与えておく。(--`Epre_defined_limit`--) は締切日を意味し、予めに(--`10`--) として宣言する。(--`Eday`--) は今日の日付を意味し、(--`5`--) と宣言する。

```
- new_open_axiom ("Epre_defined_limit", (--`Epre_defined_limit = 10`--));
val it = |- Epre_defined_limit = 10 : thm
- new_open_axiom ("Eday", (--`Eday = 5`--));
val it = |- Eday = 5 : thm
```

外部の環境を整えたので以下の定義が可能になる。

FO_CLASS ("Office","limit") とFO_CLASS ("Office","get") にに対して割り当てられた関数定義を以下に宣言する。

```
- new_definition ("limit",  (--`limit = Eday <= Epre_defined_limit`--));
val it = |- limit = Eday <= Epre_defined_limit : thm
- new_definition ("get", (--`get t:string = t`--));
val it = |- !t. get t = t : thm
```

(--`get`--) は恒等関数として定義し、(--`Ecurrentpaper`--) に反映されるものとする。構築されたモデルから HOL 上の理論に翻訳するときは、以下の作業が必要となる。

- 外部の環境は、予め定義を行っておく。
- HOL 上で定義されていない型等は実装しておく必要がある。

オブジェクト識別子の基本集合、状態識別子の基本集合、イベント識別子の基本集合、イベント識別子の基本集合を、HOL 上では型として宣言する。

```
- new_type {Arity =0, Name = "ObjectID"};
val it = () : unit
- new_type {Arity =0, Name = "StateID"};
val it = () : unit
- new_type {Arity = 0, Name = "EventID"};
val it = () : unit
- new_type {Arity = 0, Name = "EventInsID"};
val it = () : unit
```

入力イベントインスタンス集合に対して、入力イベント式がとる真偽値を求める評価関数を HOL 上で定義する。評価関数では、イベント識別子同士の比較を行うので、型コンストラクタを用いて定義する必要がある。理由は、HOL 上では項の形が異なるもの同士の比較が行えないからである。型コンストラクタを用いることによって、conversion という仕組みを用いてイベント識別子の比較を行える。これらは、値として文字列を保持させる。また、イベントインスタンス識別子に関してはそれが唯一に存在しなければならないので、値として自然数を保持させる。

```
- new_constant {Name = "EVT", Ty= (==`:string -> EventID`==)};
val it = () : unit
- new_constant {Name = "EVTINS",
                        Ty= (==`:EventID -> num -> EventInsID`==)};
val it = () : unit
```

次に、イベント式における演算子の宣言を行う。(--`EExp_AND`--) は論理和を表現する演算子であり、型を(==`:EExp_in -> EExp_in -> EExp_in`==) として定義する。(--`EExp_OR`--) についても同様である。(--`EExp_EVT`--) をイベント識別子のみをイベント式として定義する型コンストラクタとして定義する。型は(==`:EventID -> EExp_in`==) である。(--`EExp_NOT`--) は型(==`:EExp_in -> EExp_in`==) として定義し、引き数とするイベント式に対して真なら偽、偽なら真を示すものとして定義する。

(--‘EExp_EMPTY‘--) は型は(==‘:EExp_in‘==) として定義し、イベント式がないこと示す項定数である。

```
- new_infix {Name = "EExp_AND",
                Ty = (==‘:EExp_in -> EExp_in -> EExp_in‘==),Prec = 100};
val it = () : unit
- new_constant {Name = "EExp_EVT",
                Ty = (==‘:EventID -> EExp_in‘==)};
val it = () : unit
- new_infix {Name = "EExp_OR",
                Ty = (==‘:EExp_in -> EExp_in -> EExp_in‘==),Prec = 50 };
val it = () : unit
- new_constant {Name = "EExp_NOT", Ty = (==‘:EExp_in -> EExp_in‘==)};
val it = () : unit
- new_constant {Name = "EExp_EMPTY", Ty = (==‘:EExp_in‘==)};
val it = () : unit
```

入力イベントインスタンス集合に対してイベント式の取る真偽を求める評価関数(--‘Eval_eexpin‘--) を定義する。(--‘Eval_eexpin‘--) の型は(==‘:EExp_in -> EventInsID list ->bool‘==) である。これは真偽値を求める関数であるが、イベント式の各々の演算子に対して真を求める定義、偽を求める定義が必要になる。以下で、イベント式に含まれる各々の演算子に対して真偽値を求める(--‘Eval_eexpin‘--) の定義を行う。イベント式に(--‘EExp_AND‘--) を含む場合に、真を求める定義をEVT_AND_T_EVAL として宣言する。また、偽を求める定義をEVT_AND_F_EVAL として宣言する。

```
- new_open_axiom ("EVT_AND_T_EVAL",
(--‘!exp1:EExp_in.!exp2:EExp_in. !es:EventInsID list.
(Eval_eexpin:EExp_in -> EventInsID list -> bool) (exp1 EExp_AND exp2) es = (Eva
l_eexpin exp1 es) /\ (Eval_eexpin exp2 es)‘--));
val it = () : unit

- new_open_axiom ("EVT_AND_F_EVAL",
(--‘!exp1:EExp_in.!exp2:EExp_in. !es:EventInsID list.
~(Eval_eexpin (exp1 EExp_AND exp2) es) =
(~(Eval_eexpin exp1 es) /\ ~(Eval_eexpin exp2 es)) \/
(~(Eval_eexpin exp1 es) /\ (Eval_eexpin exp2 es)) \/
```

```
((Eval_eexpin exp1 es) /\ ~(Eval_eexpin exp2 es))`--));
val it = () : unit
```

上と同様に、イベント式に(--`EExp_OR`--)を含む場合に、真偽を求める定義が必要となる。真を求める定義をEVT_OR_T_EVALとして宣言する。また、偽を求める定義をEVT_OR_F_EVALとして宣言する。

```
- new_open_axiom ("EVT_OR_T_EVAL",
(--`!exp1:EExp_in.!exp2:EExp_in. !es:EventInsID list.
(Eval_eexpin:EExp_in -> EventInsID list -> bool)  (eexp1 EExp_OR exp2) es =
((Eval_eexpin exp1 es) /\ (Eval_eexpin exp2 es)) \/
(~(Eval_eexpin exp1 es) /\ (Eval_eexpin exp2 es)) \/
((Eval_eexpin exp1 es) /\ ~(Eval_eexpin exp2 es))`--));
val it = () : unit

- new_open_axiom ("EVT_OR_F_EVAL",
(--`!exp1:EExp_in.!exp2:EExp_in. !es:EventInsID list.
~(Eval_eexpin (eexp1 EExp_OR exp2) es) =
(~(Eval_eexpin exp1 es) /\ ~(Eval_eexpin exp2 es))`--));
val it = () : unit
```

イベント式が(--`EExp_NOT`--)を含む場合に、真を求める定義をEVT_NOT_T_EVALとして宣言する。また、偽を求める定義をEVT_NOT_F_EVALとして宣言する。

```
- new_open_axiom ("EVT_NOT_T_EVAL",
(--`!exp:EExp_in. !es:EventInsID list.
(Eval_eexpin (EExp_NOT exp) es) = ~(Eval_eexpin exp es)`--));
val it = () : unit

- new_open_axiom ("EVT_NOT_F_EVAL",
(--`!exp:EExp_in. !es:EventInsID list.
~(Eval_eexpin (EExp_NOT exp) es) = (Eval_eexpin exp es)`--));
val it = () : unit
```

イベント式に演算子(--`EExp_EMPTY`--)を含む場合、評価関数は必ず真を返すので以下の定義を宣言する。

```
- new_open_axiom ("EVT_EMPTY_EVAL",
(--`!es:EventInsID list.
(Eval_eexpin (EExp_EMPTY) es) = T`--));
val it = () : unit
```

イベント式がイベント識別子に対して真になる定義、偽になる定義を各々EVT_T_EVAL、EVT_F_EVAL として宣言する。

```
new_open_axiom ("EVT_T_EVAL",
(--`!e:EventID. !es:EventInsID list.
(IS_EL e (MAP EVTID es)) ==>(Eval_eexpin (EExp_EVT e)es = T)`--));
val it = () : unit

val EVT_F_EVAL =
new_open_axiom ("EVT_F_EVAL",
(--`!e:EventID. !es:EventInsID list.
~(IS_EL e (MAP EVTID es)) ==>(Eval_eexpin (EExp_EVT e) es = F)`--));
val it = () : unit
```

以上より、評価関数(--`Eval_eexpin`--) の定義が整った。これらは、入力イベントインスタンス集合に対して、真偽をもとめる証明手法が確立しているので conversion として定義できる。conversion とは、term -> thm の型を持つ関数であり、term の型を持つ項を変換して変形可能な項との等価性を表現する規則である。

次に、クラスの"Office"に属するオブジェクトの属性の状態、動作の状態を表現する関数を定義する。

- オブジェクトに対する入力イベントインスタンス集合を表現する関数(--`Office_input`--)
- オブジェクトの動作の状態を表現する関数(--`Office_state`--)
- クラス"Office"に属するオブジェクトの属性の状態を表現する関数(--`Office_env_counter`--)
- イベントインスタンスに付加する属性の状態を表現する関数(--`Office_env_list`--)

以上の各々の関数を以下の型をもつ項定数をして宣言する。

```
- new_constant {Name = "Office_input",
Ty=(==`:ObjectID -> EventInsID list`==)};
val it = () : unit
```

```
- new_constant {Name = "Office_state",Ty=(==`:ObjectID -> StateID`==)};
val it = () : unit
- new_constant {Name = "Office_env_counter",Ty=(==`:ObjectID -> num`==)};
val it = () : unit
- new_constant {Name = "Office_env_list",
Ty=(==`:ObjectID -> string list`==)};
val it = () : unit
- new_constant {Name = "Thesis_env_tname",Ty=(==`:EventInsID -> string`==)};
val it = () : unit
```

以上で、構築されたモデルの検証に必要な要素をHOL上のtheory envに定義した。作成したtheory envは以下である。

```
- print_theory "env";
Theory: env

Type constants:
    ObjectID 0
    StateID 0
    EExp_in 0
    EventID 0
    EventInsID 0

Term constants:
    Thesis_env_tname (Prefix)   :EventInsID -> string
    Office_env_list (Prefix)   :ObjectID -> string list
    Office_env_counter (Prefix)   :ObjectID -> num
    Office_input (Prefix)   :ObjectID -> EventInsID list
    Office_state (Prefix)   :ObjectID -> StateID
    Epre_defined_limit (Prefix)   :num
    Eday (Prefix)   :num
    Ecurrentpaper (Prefix)   :string
    Nil (Prefix)   :string list
    0 (Prefix)   :num
```

```
    limit (Prefix)   :bool
    update (Prefix)   :string list # string -> string list
    inc (Prefix)   :num -> num
    get (Prefix)   :string -> string
    EVT (Prefix)   :string -> EventID
    EVTINS (Prefix)   :EventID -> num -> EventInsID
    EVTID (Prefix)   :EventInsID -> EventID
    EExp_AND (Infix 100)   :EExp_in -> EExp_in -> EExp_in
    EExp_EVT (Prefix)   :EventID -> EExp_in
    EExp_OR (Infix 50)   :EExp_in -> EExp_in -> EExp_in
    EExp_NOT (Prefix)   :EExp_in -> EExp_in
    EExp_EMPTY (Prefix)   :EExp_in
    Eval_eexpin (Prefix)   :EExp_in -> EventInsID list -> bool
    s0 (Prefix)   :StateID
    s1 (Prefix)   :StateID

Axioms:
    Epre_defined_limit |- Epre_defined_limit = 10
    Eday |- Eday = 5
    EVTID |- !e n. EVTID (EVTINS e n) = e
    EVT_AND_T_EVAL
    |- !exp1 exp2 es.
         Eval_eexpin (exp1 EExp_AND exp2) es =
         Eval_eexpin exp1 es /\ Eval_eexpin exp2 es
    EVT_AND_F_EVAL
    |- !exp1 exp2 es.
         ~(Eval_eexpin (exp1 EExp_AND exp2) es) =
         ~(Eval_eexpin exp1 es) /\ ~(Eval_eexpin exp2 es) \/
         ~(Eval_eexpin exp1 es) /\ Eval_eexpin exp2 es \/
         Eval_eexpin exp1 es /\ ~(Eval_eexpin exp2 es)
    EVT_OR_T_EVAL
    |- !exp1 exp2 es.
```

```
        Eval_eexpin (eexp1 EExp_OR exp2) es =
        Eval_eexpin exp1 es /\ Eval_eexpin exp2 es \/
        ~(Eval_eexpin exp1 es) /\ Eval_eexpin exp2 es \/
        Eval_eexpin exp1 es /\ ~(Eval_eexpin exp2 es)
    EVT_OR_F_EVAL
    |- !exp1 exp2 es.
        ~(Eval_eexpin (eexp1 EExp_OR exp2) es) =
        ~(Eval_eexpin exp1 es) /\ ~(Eval_eexpin exp2 es)
    EVT_NOT_T_EVAL
    |- !exp es. Eval_eexpin (EExp_NOT exp) es = ~(Eval_eexpin exp es)
    EVT_NOT_F_EVAL
    |- !exp es. ~(Eval_eexpin (EExp_NOT exp) es) = Eval_eexpin exp es
    EVT_EMPTY_EVAL |- !es. Eval_eexpin EExp_EMPTY es = T
    EVT_T_EVAL
    |- !e es. IS_EL e (MAP EVTID es) ==> (Eval_eexpin (EExp_EVT e) es = T)
    EVT_F_EVAL
    |- !e es. ~(IS_EL e (MAP EVTID es)) ==> (Eval_eexpin (EExp_EVT e) es = F)
    EVT_EQ |- !s1 s2. (s1 = s2) = EVT s1 = EVT s2
    EVTINS_EQ
    |- !e1 e2 n1 n2. (e1 = e2) /\ (n1 = n2) = EVTINS e1 n1 = EVTINS e2 n2

Definitions:
    Nil |- Nil = []
    0 |- 0 = 0
    limit |- limit = Eday <= Epre_defined_limit
    update |- !l t. update (l,t) = CONS t l
    inc |- !n. inc n = n + 1
    get |- !t. get t = t

Theorems:
```

これまで定義した概念を基にオブジェクトの振舞いを表現する公理を定義する。

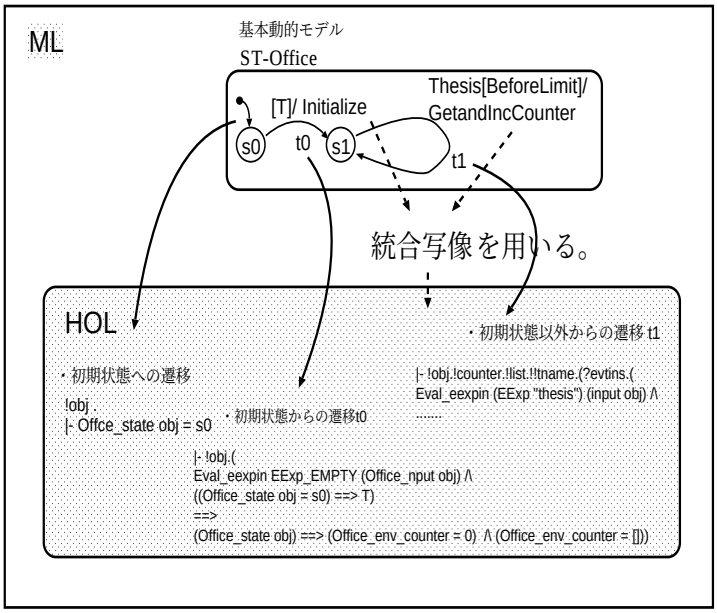


図 3.11: 状態遷移に対する公理

- 状態遷移が初期状態s0 への遷移。

  ここでは、無条件に初期状態s0 へ遷移する。したがって、オブジェクトは状態s0 となる。

公理をnew_open_axiom を用いて宣言する。

```
val axiom_init =
new_open_axiom ("axiom_init",
(--`!obj:ObjectID. Office_state obj = s0`--));
val axiom_init = |- !obj. Office_state obj = s0 : thm
```

- 状態遷移が初期状態からの遷移

状態遷移t0 の写像は状態遷移式("s0",EEXP_EMPTY,"T",,"Initialize","s1") である。ここで、状態遷移式から状態遷移前を事前条件、状態遷移後を事後条件とした公理を作成する。

"Initialize"に対するアクション式では"counter"と"list"の値が定義されていない状態から値の定義を行っているので事前条件として状態における値(--`((Office_state obj = s0) ==> T)`--) を、事後条件として(--`(Office_state obj = s1) ==> (Office_env_counter obj = 0) /\ (Office_env_list obj = [])`--) として表現する。この事前条件に、発火条件である評価関数(--`Eval_eexp_in`--) と条件

遷移式を加える。これをaxiom_t0として宣言する。ここで、(--‘((Office_state obj = s0) ==> T)‘--)の(--‘T‘--)とは、値の定義がされていないので真と定義したものである。

```
val axiom_t0 = new_open_axiom ("axiom_t0",
(--‘!obj:ObjectID.(
Eval_eexp_in EExp_EMPTY (Office_input obj) /\
((Office_state obj = s0) ==> T)
==>
(Office_state obj = s1) ==> (Office_env_counter obj = 0)
/\ (Office_env_list obj = []))‘--));
val axiom_t0 =
  |- !obj.
       Eval_eexp_in EExp_EMPTY (Office_input obj) /\
((Office_state obj = s0) ==> T)
==>
       (Office_state obj = s1) ==>
       (Office_env_counter obj = 0) /\ (Office_env_list obj = []) : thm
```

- 状態遷移 t が初期状態以外の状態からの遷移の場合

状態遷移"t1"の写像である状態遷移式("s1",EEXP_EVT "thesis","BeforeLimit","GetandIncCounter","s1") である。ここでもaxiom_t1 を宣言した時と同様に行うが、異なる点は"GetandIncCounter"の写像であるアクション式は事前条件において、値を定義できる。"GetandIncCounter"の写像を一般的な表記で示すと以下である。

```
Office.counter := Office.inc(Office.counter);Office.get(thesis.tname);
Office.list := Office.update(Office.list,thesis.tname)
```

また、この例ではイベントに付加する属性を参照している。この場合、入力イベントインスタンス集合の中にイベントインスタンス識別子の存在し、かつそのイベントインスタンス識別子に付加している属性がpre_tname であること表現する条件を事前条件に加えなければならない。

この部分は(--‘(Thesis_env_tname e = pre_tname) /\ IS_EL e (Office_input obj)‘--)と表現する。以上よりaxiom_t1 を宣言する。

```
val axiom_t1 = new_open_axiom ("axiom_t1",
(--‘!obj:ObjectID.!pre_counter:num.!pre_list:string list.!pre_tname:string.
```

```
(?e:EventInsID.(
(Eval_eexp_in (EExp_EVT (EVT "thesis")) (Office_input obj)) /\
limit /\
((Office_state obj = s1) ==>
(Office_env_counter obj = pre_counter)
/\ (Office_env_list obj = pre_list))/\
((Thesis_env_tname e = pre_tname) /\ IS_EL e (Office_input obj))))
==>
((Office_state obj = s1) ==>
(Office_env_counter obj = inc pre_counter) /\
(Office_env_list obj = update (pre_list, pre_tname)) )`--));
val axiom_t1 =
  |- !obj pre_counter pre_list pre_tname.
       (?e.
         Eval_eexp_in (EExp_EVT (EVT "thesis")) (Office_input obj) /\
         limit /\
         ((Office_state obj = s1) ==>
          (Office_env_counter obj = pre_counter) /\ (Office_env_list obj = pre_list)) /\
         (Thesis_env_tname e = pre_tname) /\
         IS_EL e (Office_input obj)) ==>
       (Office_state obj = s1) ==>
       (Office_env_counter obj = inc pre_counter) /\
       (Office_env_list obj = update (pre_list,pre_tname)) : thm
```

ここで、生成された公理は事前条件と事後条件を用いて定義しており、事前条件をみたす定理の集合に対して、事後条件をみたす定理を生成する規則を定義できる。これは、MP を用いて定義できる。

事後条件をみたす定理とは、オブジェクトの状態における属性を示す定理を意味しており、この規則を用いることにより、目的とする性質を容易に調べることが可能となる。以下に規則を定義する。

axiom_t1 に対する規則は、対象とするオブジェクトである項、オブジェクトに対する入力イベントインスタンス集合を表現する定理をとり、状態遷移t1 が起きた後の対象とするオブジェクトの状態における属性の値を表現する定理を返す。この規則を、TRANSITION_T0 として定義する。この規則に関しては後に詳細に説明する。

```
val TRANSITION_T0 = fn : term -> thm -> thm
```

axiom_t2 に対する規則は、対象とするオブジェクトである項、オブジェクトの状態における属性の値を表現する定理、オブジェクトに対する入力イベントインスタンス集合を表現する定理、イベントインスタンスに付加する属性の値を表現する定理、オブジェクトの発火条件を満たした定理、オブジェクトの状態における属性の値を表現する定理をとり、状態遷移 t2 が起きた後の対象とするオブジェクトの状態における属性の値を表現する定理を返す。この規則を、TRANSITION_t1 として定義する。

```
val TRANSITION_T1 = fn : term -> thm -> thm -> thm -> thm -> thm
```

以上により検証を行うための環境は整った。

3.5.2 属性の値に関する性質の証明

ここで、例題においてはクラス"Office"のインスタンスである(--‘office1‘--)が状態(--‘s1‘--)において属性"counter" の値がある特定の値になるかについて検証を行う。また、属性"counter"の値として(--‘2‘--)を考える。ここで行う検証は入力イベントインスタンス集合を定理として与える。また、遷移条件については様々な型などで実装されるので、手作業で証明を行う。証明すべき命題prop は以下である。

```
- val prop = (--‘(Office_state office1 = s1) ==> (Office_env_counter office1 = 2) ‘--);
val prop = ‘‘(Office_state office1 = s1) ==> (Office_env_counter office1 = 2)‘‘ : term
```

new_constant を用いてオブジェクト office1 を宣言する。また、mk_thm を用いて、office1 の入力イベントインスタンス集合を表現する定理を作成する。ここでは、入力イベントインスタンス集合を表現する定理を新しい axiom または新しい definition としてtheory env に加えないものとする。

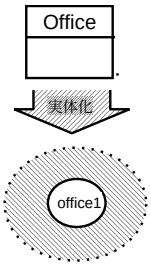


図 3.13: 初期状態

実体化されたオブジェクト"office1"は、以下の図 3.13のような動作の状態をもつ。

次に、"office1"に与える入力イベントインスタンス集合を定義する。ここでは、空集合とする。

```
- val input0 =
mk_thm ([],(--`Office_input office1 = ([]:EventInsID list)`--));
- val input0 = |- Office_input office1 = [] : thm
```

入力イベントインスタンス集合を表現するinput0を用いて、状態遷移"t0"に関して定義した規則TRANSITION_T0を適用し、証明された定理をenv1とする。

```
- val env1 = TRANSITION_T0 (--`office1`--) input0;
val env1 =
  |- (Office_state office1 = s1) ==>
     (Office_env_counter office1 = 0) /\ (Office_env_list office1 = []) : thm
```

ここで、TRANSITION_T0について説明する。oid、input0、axiom_t0は以下になっている。

```
- oid;
val it = ``office1`` : term
- input0;
val it = |- Office_input office1 = [] : thm
- axiom_t0;
val it =
  |- !obj.
       Eval_eexpin EExp_EMPTY (Office_input obj) /\
       ((Office_state obj = s0) ==> T) ==>
       (Office_state obj = s1) ==>
       (Office_env_counter obj = 0) /\ (Office_env_list obj = []) : thm
```

始めに、axiom_t0 の(--`obj`--) をSPEC を用いて、(--`office1`--) に置き換える。次に、input0 のイコールの右をとり、(--`Eval_eexpin (EExp_EMPTY)`--) に連結させる。

```
- val th1 = SPEC oid axiom_t0;
val th1 =
  |- Eval_eexpin EExp_EMPTY (Office_input office1) /\
     ((Office_state office1 = s0) ==> T) ==>
     (Office_state office1 = s1) ==>
     (Office_env_counter office1 = 0) /\ (Office_env_list office1 = []) : thm
- val prop = mk_comb {Rator = (--`Eval_eexpin (EExp_EMPTY)`--),
Rand = (#rhs (dest_eq (#2 (dest_thm input0))))};
val prop = ``Eval_eexpin EExp_EMPTY []`` : term
```

次に、評価関数(--`Eval_eexpin`--) に対し、作成した convertion であるEEXP_EVAL_CONV を用いて真を導く。lemma1 をinput0 を用いて書換える。

```
- val lemma1 = EEXP_EVAL_CONV prop;
val lemma1 = |- Eval_eexpin EExp_EMPTY [] = T : thm

- val th2 = EQT_ELIM (PURE_REWRITE_RULE [(SYM input0)] lemma1);
val th2 = |- Eval_eexpin EExp_EMPTY (Office_input office1) : thm
```

axiom_init の(--`obj`--) をSPEC を用いて、(--`office1`--) に置き換え、EQT_INTRO を用いて、(--`T`--) と等価である定理を導き、イコールを含意に変換する。

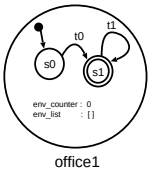
```
- val th3 = #1 (EQ_IMP_RULE (EQT_INTRO (SPEC oid axiom_init)));
val th3 = |- (Office_state office1 = s0) ==> T : thm
```

ここで、th2、th3 にCONJ を適用する。次にMP を用いて目的とする定理となる。

```
- val th4 = CONJ th2 th3;
val th4 =
  |- Eval_eexpin EExp_EMPTY (Office_input office1) /\
     ((Office_state office1 = s0) ==> T) : thm
- MP th1 th4;
val it =
  |- (Office_state office1 = s1) ==>
     (Office_env_counter office1 = 0) /\ (Office_env_list office1 = []) : thm
```

この一連の証明を行う規則TRANSITION_T0 は以下である。

```
fun TRANSITION_T0 oid th_input =
let
val th1 = SPEC oid axiom_t0
        val prop = mk_comb {Rator = (--`Eval_eexpin (EExp_EMPTY)`--),
    Rand = (#rhs (dest_eq (#2 (dest_thm th_input))))}
  val lemma1 = EEXP_EVAL_CONV prop
        val th2 = EQT_ELIM (PURE_REWRITE_RULE [(SYM th_input)] lemma1)
val th3 = #1 (EQ_IMP_RULE (EQT_INTRO (SPEC oid axiom_init)))
in
MP th1 (CONJ th2 th3)
end;
```



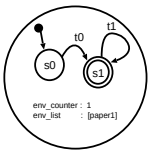
```
(REWRITE_TAC(REWRITE_FUNC_BASE@REWRITE_ENV_BASE))
THEN
(CONV_TAC ARITH_CONV));
val cond1 = |- limit : thm

- val input1 = mk_thm ([],
(--`Office_input office1 = [(EVTINS (EVT "thesis") 1)]`--));
val input1 = |- Office_input office1 = [EVTINS (EVT "thesis") 1] : thm

- val evt_env1 = mk_thm ([],
(--`Thesis_env_tname (EVTINS (EVT "thesis") 1) = "paper1"`--));
val evt_env1 = |- Thesis_env_tname (EVTINS (EVT "thesis") 1) = "paper1" : thm
```

入力イベントインスタンス集合を表現する定理、遷移条件の充足を表現する定理、入力イベントインスタンスに付加している属性の値を表現する定理を状態遷移t1 に関して定義した規則TRANSITION_T1 を適用する。次に、関数として定義した(--`inc`--) 等と arithmetic theory を用いて、書換える。

```
- val env2 = TRANSITION_T1 (--`office1`--) env1 evt_env1 cond1 input1;
val env2 =
  |- (Office_state office1 = s1) ==>
     (Office_env_counter office1 = inc 0) /\ (Office_env_list office1 = update ([],"paper1"))
  : thm
```



office1

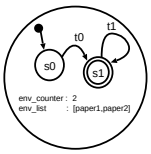
同様に各々の定理を導くまたは証明を行って、規則TRANSITION_T1 を適用する。

```
- val cond2 = prove((--`limit`--),
(REWRITE_TAC(REWRITE_FUNC_BASE@REWRITE_ENV_BASE))
THEN
(CONV_TAC ARITH_CONV));
val cond2 = |- limit : thm

- val input2 = mk_thm ([],
(--`Office_input office1 = [(EVTINS (EVT "thesis") 2)]`--));
val input1 = |- Office_input office1 = [EVTINS (EVT "thesis") 2] : thm

- val evt_env2 = mk_thm ([],
(--`Thesis_env_tname (EVTINS (EVT "thesis") 2) = "paper2"`--));
val evt_env2 = |- Thesis_env_tname (EVTINS (EVT "thesis") 2) = "paper1" : thm

- val env4 = TRANSITION_T1 (--`office1`--) env3 evt_env2 cond2 input2;
val env4 =
  |- (Office_state office1 = s1) ==>
     (Office_env_counter office1 = inc 1) /\
     (Office_env_list office1 = update (["paper1"],"paper2")) : thm
```



office1

```
- hd(IMP_CANON (REWRITE_RULE [EQT_ELIM (ARITH_CONV (--`1 + 1 = 2`--))] env5));
val it = |- (Office_state office1 = s1) ==> (Office_env_counter office1 = 2) : thm
```

以上で命題が証明された。

第 4 章

検証フレームワーク

前章までに、各々の検証法に対してどのようにして検証支援が行えるかを示した。Well-formedness チェック、データの流れに注目した一貫性検証に関しては自動証明を行う手続きを予めライブラリとして実装しておき、検証を行いたいときに、そのライブラリを用いて適用できるようにする。これは、ML を用いて行う。HOL を用いておこなった検証法に対しては、各々の検証法に対して検証アプリケーションを作成しなければならない。特に、アプリケーションドメインに依存した検証に関しては、モデルの構築が進むにつれて検証すべき事項が洗い出され、検証の種類も無数に存在する。

そこで、検証フレームワークという概念を提案する。検証フレームワークは、形式的モデルに基づいた検証を行う為の枠組みである。また、検証フレームワークでは、検証アプリケーションをフローズンスポットとホットスポットに分解する。フローズンスポットは複数の検証アプリケーションに共通な部分であり、あらかじめ実装しておく。ホットスポットは検証アプリケーション毎に異なる部分である。検証アプリケーションを作成する場合はホットスポットの部分だけを実装して実現できる。これにより、検証アプリケーション作成のコストが削減され、柔軟に計算機上で検証を行うことが可能となる。この章では、検証フレームワークの概念を説明する。

4.1 検証フレームワーク

検証フレームワークでは、モデル構築情報が ML の参照型を用いて保持されている。HOL を用いて支援を行う検証法では、モデル情報に対して加工/選択したりする手続き的な処理が必要である。そのような作業は、モデル情報にアクセスするプリミティブを用いて行う。よって、検証アプリケーションでは、そのアクセスするプリミティブを用いて加工/選択を行い、証明を行う環境を生成することになる。また、様々な検証法を扱うためには、HOL 上の証明を行う環境を生成する部分を柔軟に変更できるようにする必要がある。そこで、割り当て関数、翻訳関数、生成関数を用いた生成法を定義する。以下にその概要を示す。

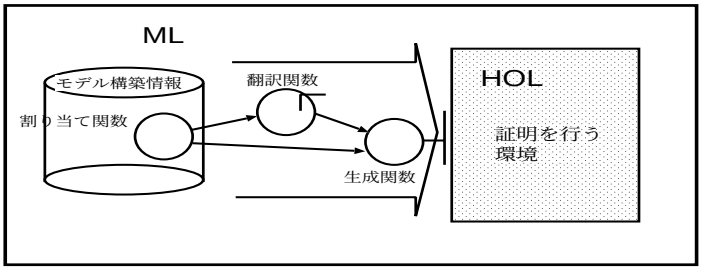


図 4.1: 検証フレームワークを用いた検証アプリケーション

モデル構築情報から HOL 上の世界を構築するには、以下の割り当て関数、翻訳関数、生成関数を用いる。

- 割り当て関数
  モデルの構成要素の HOL 上で取り扱われる概念に割り当てる。
- 翻訳関数
  HOL 上の環境のもとにおける解釈を与える。
- 生成関数
  割り当てとモデルの意味に基づいて、HOL 上の環境を生成する。

構築したモデルに対して割り当てる概念、つまり与える情報の変更に対して、柔軟に対処できるようにしなければならない。これを割り当て関数として定義する。また、割り当て関数によって、与えられた詳細な情報と必要とすることなく独立して実現できる部分、これを翻訳関数として定義する。また、検証法によって、証明を行う環境は異なる。これに柔軟に対象する部分が生成関数である。

次に、どの検証法に関しても共通した部分、つまり形式的モデルに固有な部分は予め実装を行うことができる。ここで、形式的モデルに固有な部分の割り当て関数、翻訳関数、生成関数は、３つの関数が１かたまりである特定の HOL 上の世界を生成するものである。したがって、異なる HOL 上の世界を生成するものを別のモジュールとして実装する。

検証フレームワークの機能を洗い出すと以下になる。

- 参照型に格納されたモデル情報。
- モデル構築情報に対して、アクセスするプリミティブ。
  このプリミティブを用いることにより、モデルの加工、選択といった作業が必要になる検証法に対して有効である。
- ある特定の世界観を構築できる３つの関数 (割り当て関数、 翻訳関数、生成関数) が用意されている。

4.2 割り当て関数と翻訳関数

適用した例題に関して構築された世界観を取り上げる。例題に関しては以下の２つの世界観を構成できるモジュール化された割り当て関数、翻訳関数が定義される。

4.2.1 不変表明を用いた一貫性検証で構築された世界観

例題で構築された世界観の割り当て関数、翻訳関数を示す。

属性識別子に対する割り当て関数は`AO * string * hol_type -> unit`の型をもつ`bind_type`によって実現される。、引数はそれぞれ属性識別子、HOL の世界における名前、型の順である。例題では、属性識別子`"counter"`、`"list"`、`"tname"`は割り当て関数を用いて、以下のように割り当てられる。

```
bind_type (AO_CLASS("Office","counter"),"counter",(==`:num`==));
bind_type (AO_CLASS("Office","list"),"list",(==`:string list`==));
bind_type (AO_EVT("thesis","tname"),"tname",(==`:string`==));
```

関数識別子に対する割り当て関数は`FO * string * string -> unit`の型をもつ`bind_def`によって表現される。引数はそれぞれ関数識別子、対象とする関数識別子、HOL で割り当てる名前、そしてその定義である。例題では、関数識別子`"inc"`、`"update"`、`"limit"`は割り当て関数を用いて、以下のように割り当てられる。

```
bind_def (FO_CLASS("Office","inc"), "inc", "inc n:num = n + 1");
bind_def (FO_CLASS("Office","update"),"update",
        "update ((l:string list),(t:string)) = CONS t l");
bind_def (FO_CLASS("Office","limit"), "limit",
                        "limit = Eday <= Epre_defined_limit");
```

外部の環境に影響する関数識別子に対する割り当て関数を導入した。この関数は`FO * string * string * string -> uni`の型をもつ`bind_sideeffect`によって表現される。引数はそれぞれ関数識別子、対象とする関数識別子、HOL で割り当てる名前、影響を与える外部の環境、そしてその定義である。

```
bind_def(FO_CLASS("Office","O"),"O", "O = 0");
bind_def(FO_CLASS("Office","Nil"),"Nil", "Nil = ([]:string list)");

bind_sideeffect (FO_CLASS("Office", "get"),
        "get", "Ecurrentpaper", "get t:string = t");
```

以上により定義された各々の HOL の概念を用いた表現は、属性識別子に対してmake_variables、関数識別子に対してmake_definition により HOL の世界を構築できる。

例題ではアクション式AExp、論理式BExp に対しての翻訳関数が定義されている。論理式BExp に対しては(==‘bool‘==) を返す関数tl_bexp、アクション式AExp に関しては{redex:term,residue:term} list を返す関数tl_aexp である。以下に詳しく述べる。アクション式に関する翻訳関数tl_aexp は型はAExp -> {redex:term,residue であり、これは、アクション式を項定数の置換として解釈した関数である。{redex:term, residue:term}は ML 関数val subst: {redex:term, residue:term} -> term -> term によって 2 つ目の引数に対して置換をおこなった項を返す。例えば、subst [{redex = (--‘SUC 0‘--),residue = (--‘1‘--)}](--‘SUC0‘--) は(--‘1‘--) を返す関数である。例題では以下の 2 つがある。

```
Office.counter := 0 ;Office.list := []
```

```
Office.counter := Office.inc(Office.counter);Office.get(thesis.tname);
Office.list := Office.update(Office.list,thesis.tname)
```

このAExp に対して、翻訳関数tl_aexp は以下である。

```
- tl_aexp (AEXP_CONS
       (AEXP_SUBST
           (AO_CLASS ("Office","counter"),
            TERM_FUNC (FO_CLASS ("Office","O"),[])),
         AEXP_SUBST
           (AO_CLASS ("Office","list"),
            TERM_FUNC (FO_CLASS ("Office","Nil"),[]))));
val it = [{redex=``counter``,residue=``O``},{redex=``list``,residue=``Nil``}]
  : {redex:term, residue:term} list

- tl_aexp (AEXP_CONS  (
       AEXP_CONS  (
          AEXP_SUBST (AO_CLASS("Office","counter"),
            TERM_FUNC (FO_CLASS("Office","inc"),
                        [TERM_ATTR (AO_CLASS("Office","counter"))])),
          AEXP_APP (FO_CLASS("Office","get"),
                        [TERM_ATTR (AO_EVT("thesis","tname"))])),
       AEXP_SUBST (AO_CLASS("Office","list"),
```

```
          TERM_FUNC (FO_CLASS("Office","update"),
                        [TERM_ATTR (AO_CLASS("Office","list")),
                                TERM_ATTR
(AO_EVT("thesis","tname"))]))));
val it =
  [{redex=``counter``,residue=``inc counter``},
   {redex=``Ecurrentpaper``,residue=``get tname``},
   {redex=``list``,residue=``update (list,tname)``}]
  : {redex:term, residue:term} list
```

論理式BExp に対する翻訳関数は以下である。

```
- tl_bexp (BEXP_APP ( FO_CLASS ("Office","limit"),[]));
val it = ``limit`` : term
```

4.2.2 状態に注目した属性の値に関する検証で構築された世界観

ここでも同様に構築された世界観の割り当て関数、翻訳関数を示す。この関数は異なるモジュールで実装されているので上と同じ関数を用いて表現する。

属性識別子に対する割り当て関数はAO * string * hol_type -> unit の型をもつbind_type によって実現される。、引数はそれぞれ属性識別子、HOL の世界における名前、型の順である。例題では、属性識別子"counter"、"list"、"tname"は割り当て関数を用いて、以下のように割り当てられる。不変表明を用いた一慣性検証と同じ型であるが、ここでの割り当てた HOL 上の項は変数として使われる。

```
bind_type (AO_CLASS("Office","counter"),"counter",(==`:num`==));
bind_type (AO_CLASS("Office","list"),"list",(==`:string list`==));
bind_type (AO_EVT("thesis","tname"),"tname",(==`:string`==));
```

関数識別子に対する割り当て関数はFO * string * string -> unit の型をもつbind_def によって表現される。引数はそれぞれ関数識別子、対象とする関数識別子、HOL で割り当てる名前、そしてその定義である。例題では、関数識別子"inc"、"update"、"limit"は割り当て関数を用いて、以下のように割り当てられる。

```
bind_def (FO_CLASS("Office","inc"), "inc", "inc n:num = n + 1");
bind_def (FO_CLASS("Office","update"),"update",
        "update ((l:string list),(t:string)) = CONS t l");
bind_def (FO_CLASS("Office","limit"), "limit",
```

```
                    "limit = Eday <= Epre_defined_limit");
```

外部の環境に影響する関数識別子に対する割り当て関数を導入した。
この関数はFO * string * string * string -> unit の型をもつbind_sideeffect によって表現される。引数はそれぞれ関数識別子、対象とする関数識別子、HOL で割り当てる名前、影響を与える外部の環境、そしてその定義である。

```
bind_def(FO_CLASS("Office","O"),"O", "O = 0");
bind_def(FO_CLASS("Office","Nil"),"Nil", "Nil = ([]:string list)");

bind_sideeffect (FO_CLASS("Office", "get"),
        "get", "Ecurrentpaper", "get t:string = t");
```

以上により定義された各々の HOL の概念を用いた表現は、属性識別子に対してmake_variables、関数識別子に対してmake_definition により HOL の世界を構築できる。

例題ではアクション式AExp、論理式BExp、イベント式EExp_in に対しての翻訳関数が定義されている。論理式BExp に対しては(==‘bool‘==) を返す関数tl_bexp、アクション式AExp に関してはterm を返す関数tl_aexp である。

アクション式に関する翻訳関数tl_aexp は型はAExp -> term であり、これは、アクション式を事前条件と事後条件を用いた解釈を行う。イベント式に対しての翻訳関数tl_eexpin はイベントインスタンス集合に対して、イベント式が真偽値を求める評価関数として翻訳する。

```
- tl_aexp (AEXP_CONS
        (AEXP_SUBST
            (AO_CLASS ("Office","counter"),
             TERM_FUNC (FO_CLASS ("Office","O"),[])),
          AEXP_SUBST
            (AO_CLASS ("Office","list"),
             TERM_FUNC (FO_CLASS ("Office","Nil"),[]))));
val it = ``!obj. T ==>
        ((Office_env_counter obj = O) /\ (Office_env_list obj = Nil))``: term

- tl_aexp (AEXP_CONS  (
       AEXP_CONS  (
```

```
          AEXP_SUBST (AO_CLASS("Office","counter"),
            TERM_FUNC (FO_CLASS("Office","inc"),
                        [TERM_ATTR (AO_CLASS("Office","counter"))])),
          AEXP_APP (FO_CLASS("Office","get"),
                        [TERM_ATTR (AO_EVT("thesis","tname"))])),
       AEXP_SUBST (AO_CLASS("Office","list"),
          TERM_FUNC (FO_CLASS("Office","update"),
                        [TERM_ATTR (AO_CLASS("Office","list")),
                                TERM_ATTR
(AO_EVT("thesis","tname"))]))));
val it = ``!obj counter list tname.
((Office_env_counter obj = counter) /\ (Office_env_listobj = list))
==>
((Office_env_counter obj = inc counter) /\
        (Office_env_list obj = update (list,tname)))``
  : term

- tl_bexp (BEXP_APP ( FO_CLASS ("Office","limit"),[]));
val it = ``limit`` : term

- tl_eexpin(EEXP_EVT "thesis");
val it = ``Eval_eexpin (EExp_EVT (EVT " thesis "))`` : term
```

4.3 検証フレームワークを用いた検証アプリケーションの作成

前章において、検証フレームワーク上では、ある特定の世界観に基づいた翻訳関数、割り当て関数が定義されている。これらは検証法に非依存の部分であり、検証アプリケーションにおけるフローズンスポットとなる。検証アプリケーションは、検証法依存の部分であるホットスポットを実装することにより、容易に作成できる。これらはモデルに依存していているので予め実装しておくことは不可能であり、検証フレームワークを用いた検証アプリケーション作成のアーキテクチャは以下となる。

4.3.1 不変表明を用いた一貫性検証

不変表明を用いた一貫性検証の例において検証アプリケーション作成を行う。ホットスポットとして、不変表明をクラスに対して割り当て、クラスに存在するすべての不変表明を充足する命題を示さなければならない。ここで命題を生成する翻訳関数tl_term を定義する。しかし、不変表明を用いた検証は普遍的な性質をもっているので、これらに関してはライブラリして実装されているものである。

クラスの不変表明の割り当て関数bind_inv を用いて、割り当てる。

```
bind_inv ("Office","counter >= 0 /\ (LENGTH list = counter)");
```

また、翻訳関数を追加する。追加する翻訳関数は"Office"を引数にとり、割り当てられた不変表明を基に各々の状態遷移に対しての証明すべき命題を生成する関数である。

```
- tl_inv "Office";
val it =
  [``T /\ T ==> 0 >= 0 /\ (LENGTH Nil = 0)``,
   ``(counter >= 0 /\ (LENGTH list = counter)) /\ limit ==>
     inc counter >= 0 /\ (LENGTH (update (list,tname)) = inc counter)``]
  : term list
```

ここで、以下の作業をおこない、make_world を作成する。

1. 外部の環境を"initialized.hol"ファイルに書きこむ。
2. make_variables を実行。
3. make_definition を実行。
4. tl_inv をすべてのクラスに適用し、その全ての命題をの証明を行う。

4.3.2 状態に注目した属性の値に関する検証

ホットスポットとしては、状態遷移に対して HOL 上の概念である公理と対応づけられる翻訳する関数を追加する。tl_std_trans を定義する必要がある。これは、状態遷移図の全ての遷移に対して公理を作成する関数である。

ホットスポットとして追加する関数tl_std_trans は、 "ST_Office"に対して適用すると以下である。

```
- tl_std_trans "ST_Office";
val it =
```

```
[("init",``!obj. Office_state obj = s0``),
 ("t0",
  ``!obj.
      Eval_eexpin EExp_EMPTY (Office_input obj) /\ T /\ ((Office_state obj = s0) ==> T) ==>
      (Office_state obj = s1) ==>
      (Office_env_counter obj = 0) /\ (Office_env_list obj = Nil)``),
 ("t1",
  ``!obj counter list tname.
      (?e1.
        Eval_eexpin (EExp_EVT (EVT " thesis ")) (Office_input obj) /\
        limit /\
        ((Office_state obj = s1) ==>
         (Office_env_counter obj = counter) /\ (Office_env_list obj = list)) /\
        (Thesis_env_tname e1 = tname) /\
        IS_EL e1 (Office_input obj)) ==>
      (Office_state obj = s1) ==>
      (Office_env_counter obj = inc counter) /\
      (Office_env_list obj = update (list,tname))``)] : (string * term) list
```

不変表明を用いた一貫性検証と同様に make_world の作成を行う。

1. 外部の環境を"initialized.hol"ファイルに書きこむ。
2. make_variables を実行。
3. make_definition を実行。
4. make_type を実行。
5. 検証を行いたいオブジェクトと対応する状態遷移図に対して、tl_std_trans を適用し、証明を行う。

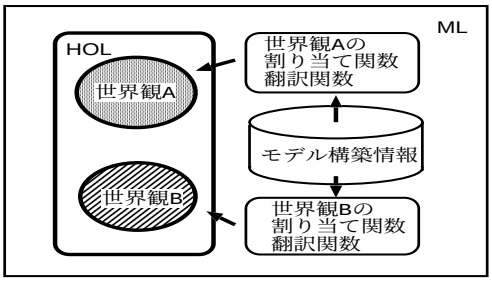


図 4.2: 世界の異なる割り当て関数、翻訳関数、生成関数

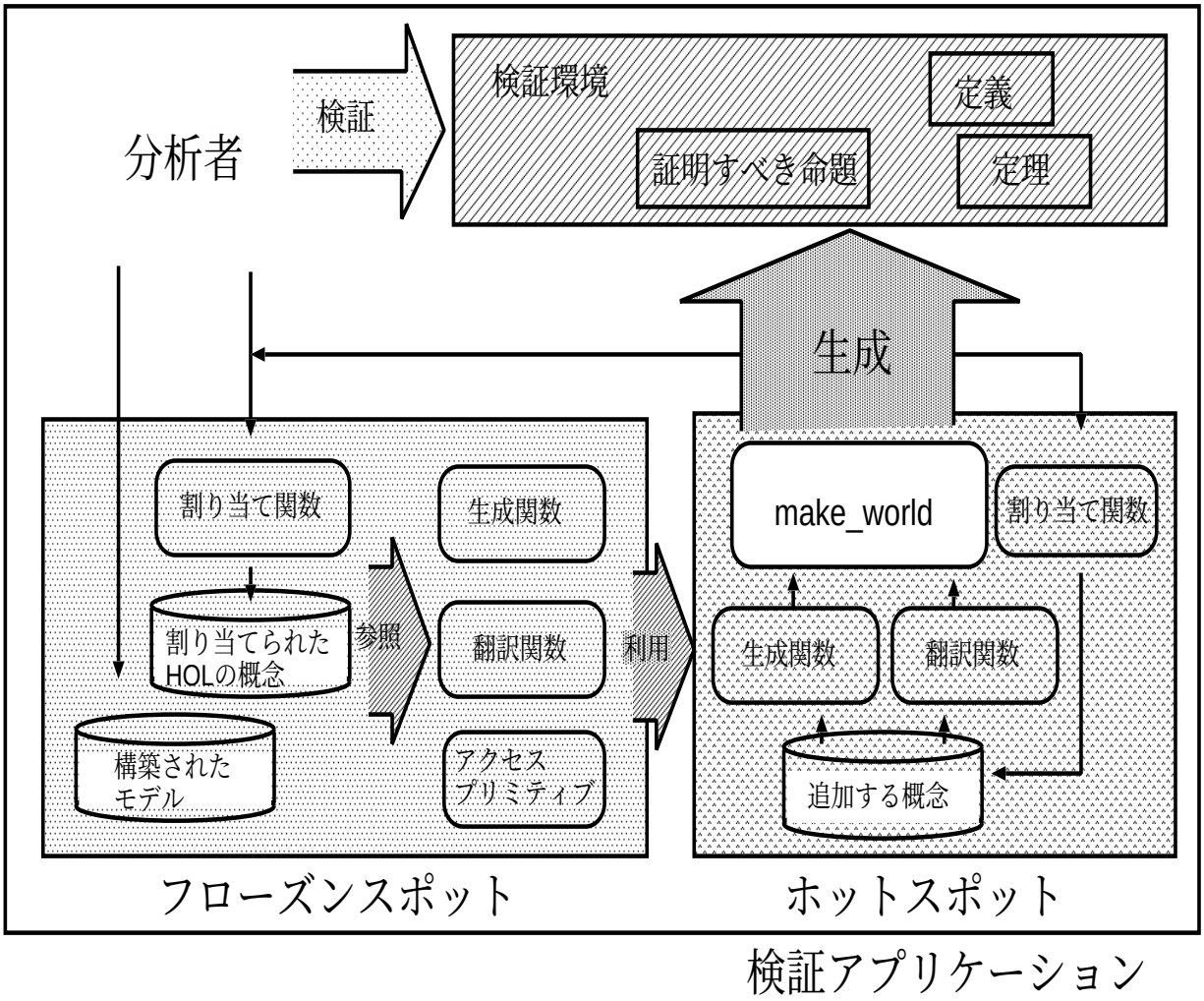


図 4.3: 検証アプリケーション作成のアーキテクチャ

第 5 章

おわりに

5.1 まとめ

本研究では、既に提案されている形式的モデルを ML 上に実装した。次に、形式的モデルに基づいた各々の検証法に対し、HOL を用いて検証環境を作成し検証を行った。検証アプリケーションに対して、抽出されたフロースポットを検証フレームワーク上で実装した。検証フレームワークを用いた検証アプリケーションの作成の手順を明確にした。

5.2 今後の課題

本研究では、3つの検証法を例題に適用し、フローズンスポットを抽出していったが、もっと多くの検証法に対し進めることでより洗練されたフレームワークになると考えられる。検証フレームワークに対する要求として挙げられる事項は、イベント通信のメカニズム、リンクの実体化などがある。また、大規模なシステム開発においての検証フレームワークの有効性を調べていないので、その評価を行う。

謝辞

本研究を行うに当たり、終始御指導頂きました片山卓也教授には、心からの感謝を申し上げます。また、本研究に関しての助言や多くの有意義な後意見を頂きました権藤克彦助教授、伊藤恵助手、Adel Cherif 助手、青木利晃氏ならびに片山研究室の皆様に厚くお礼申し上げます。

参考文献

[1] 青木利晃: オブジェクト指向方法論のための形式的モデル, Master's thesis, JAIST, 1996.

[2] 青木利晃, 石田至, 古川順一, 片山卓也: オブジェクト指向分析モデルにおける一貫性検証のための公理系の実装, ソフトウェア科学会 第 14 回全国大会,pp 465-468 , 1997.

[3] 石田至: オブジェクト指向方法論のための形式的モデルの検証, Master's thesis, JAIST, 1997.

[4] Ramgaugh,J.Blaha,M.,Premerlani,M.,Eddy,F. and Lorensen,W.: Obect-Oriented modeling and design, Prentice-Hall International, 1991.

[5] M.J.C.Gordon, T.F.Melham: Introduction to HOL CAMBRIDGE UNIVERSITY PRESS, 1993

[6] Judy,Crow,Sam Owre, John Rushby, Natarajan Shankar,Mandayam Srivas: A Tutorial Introduction to PVS, WIFT'95, 1995