ソフトウェア工学 122−14
（1999. 3. 19）

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

花田真樹† 青木利晃† 片山卓也†

現在、多くのオブジェクト指向方法論が提案されている。これらで用いられるモデルは、形式化が十分でないので計算機による意味的支援を十分に行えない。この問題点を解決するために、我々はオブジェクト指向方法論のための形式的モデルを提案している。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。手法として、検証フレームワークという概念を用いる。検証フレームワークは、構築されたモデルからHOLの概念へ変換する関数を予め与えたものであり、この関数を用いて検証法依存の部分を実装することにより容易に検証環境を作成することが可能となる。

Verification Framework for Object-Oriented Methodology

MASAKI HANADA,† TOSHIAKI AOKI† and TAKUYA KATAYAMA†

Today many object-oriented methodologies are proposed. Since models used in these methodologies are not formalized, it is difficult to support checking semantic with computers. To solve this problem, We proposed *formal models* for object-oriented methodologies. In this paper, We propose a computer-supported method for verifications based on the fomal model. We adopt *verification framework*. In verification framework, We implement functions that transform built models to objects on HOL. These functions make it easier to implement verification environments.

1. はじめに

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[1]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、対象システムをほぼ直交する3つの視点に分割して記述する。しかし、形式化が十分でないので、これらのモデルでは、計算機による意味的支援が十分に行えない。この問題点を解決するために、我々はオブジェクト指向方法論のための**形式的モデル**[2)~4)] (以下、形式的モデル)を提案している。形式的モデルでは、OMT法で用いられる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ている。

特定の検証法を計算機上で支援するソフトウェアのことを我々は**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などがある。

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、各々の検証法に対する検証アプリケーションの作成コストを削減し、有用な検証支援環境を構築することを目的とする。検証フレームワーク上には、基礎となる関数を与える。検証アプリケーションの作成は検証法に依存したホットスポットだけを実装すればよいので、検証支援環境の実現が容易になる。本論文では、まず、形式的モデルの概要を説明し、不変表明を用いた一貫性検証を紹介する。そして、検証フレームワークを提案し、例題に関して、目的とする検証アプリケーションが容易に作成できることを明らかにする。

2. 形式的モデル

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

† 北陸先端科学技術大学院大学 情報科学研究科
School of Information Science, Japan Advanced Institute of Science and Technology

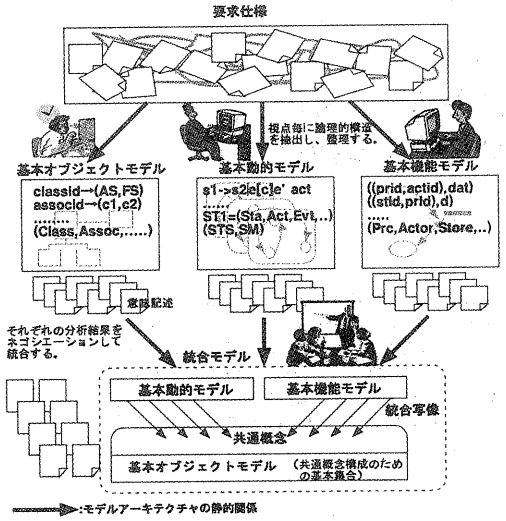

図1　形式的モデルの概念

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。形式的モデルの概念は図1に示す。検証支援環境では、これらの構築されたモデルはML上の参照型に格納する。

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する。モデルの基本集合として各々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ている。識別子は各々の側面に固有な概念を抽象化しているため、これらの持つ意味は別にドキュメイント化する。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。基本モデルでは、基本集合を文字列型、集合をリスト型を用いて定義する。形式的モデルの説明にMLのデータ型を用いる。

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する。

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`により表現する。この対応関係は写像`MO_class`を用いて表現する。

```
type ClassID = string;
type ClassExp = AttrID list * FuncID list;
type MO_class = (ClassID * ClassExp) list;
```

関連、集約関係、継承関係も同様に、出現する識別子とそれらが示す内容を対応付ける。

```
type AssocID = string;
type AssocExp = (ClassID * LimitID * MExp)
              * (ClassID * LimitID * MExp)
type MO_assoc = (AssocID * AssocExp) list
type AggrID = string;
type AggrExp = ClassID *
                    (ClassID * MExp) list;
type MO_aggr = (AggrID * AggrExp) list;
type InherID = string;
type InherExp = ClassID * ClassID list;
type MO_inher = (InherID * InherExp) list;
```

基本オブジェクトモデル`ObjectModel`は、対象システムに出現する各識別子集合と、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する。

```
type ObjectModel = AttrID list * FuncID list
* ClassID list * AssocID list * AggrID list
* InherID list * MO;
type MO = MO_class * MO_lprop * MO_assoc *
                          MO_aggr * MO_inher;
```

2.1.2　基本動的モデル

基本動的モデル`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`を用いて定義する。

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

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、状態遷移図に出現す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、状態遷移識別子の内容を表現する状態遷移式`TransExp`によって定義する。この対応関係は写像`MD_trans`を用いて定義する。

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

状態遷移式`TransExp`が`("s1",EExp_in"eventin","cond",["eventout"],"action","s2")`である場合、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`"eventout"`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。

2.1.3　基本機能モデル

基本機能モデル`FunctionModel`は、対象システムに出現するプロセス識別子の集合、アクター識別子の

集合、ストア識別子の集合、データストア識別子の集合、データフロー識別子の集合と各々のデータフロー識別子が示す内容を対応づける写像MF_flowによって定義する。

```
type FunctionModel = ProcessID list *
                     ActorID list * StoreID list
                       * FlowID list * MF_flow;
```

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によって定義する。対応関係は写像MF_flowで定義する。

```
type FlowID = string;
type FlowExp = (Node * Node) * DataID;
type MF_flow = (FlowID * FlowExp) list;
```

ノードはプロセス、アクター、データストアのいずれかである。

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

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つの統合モデルを構成できる。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行う。

論文受理システム (図2) を用いて、主な統合写像を説明する。分析モデルとして、形式的モデルの図による表記を図3に示す。

統合写像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。例題では、"Student"と状態遷移図"ST_Student"を対応づける。"Office"も同様に状態遷移図"ST_Office"と対応づける。

```
[("Student","ST_Student"),
    ("Office","ST_Office")]:UST;
```

統合写像Ucは遷移条件の統合写像であり、遷移条件識別子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状態遷移論理式BExpを対応付ける写像である。状態遷移論理式BExpはML上ではデータ型を用いて定義しているが、ここでは一般的な記法を用いる。

```
[("T",true),
 ("BeforeLimit",Office.limit())
 ("NotFinish",Student.finish()),
 ("Finish",Student.finish())]:Uc
```

統合写像Uaはアクションの統合写像であり、アクション識別子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アクション式AExpを対応付ける写像である。アクション式AExpはML上ではデータ型を用いて定義しているがここでは一般的な記法を用いる。

```
[("Initialize",
Office.counter:=0;Office.list=[]),
("GetandInitCounter",
Office.counter:=Office.inc(Office.counter);
Office.get(thesis.tname);
Office.list:=Office.update(Office.list,
                           thesis.tname)]:Uc;
("WriteThesis",Student.tname:=Student.write())
("ReleaseThesis",thesis.tname:=Student.tname)
```

この他に例題ではイベントへの属性付加の統合写像Ue、イベント出力先の統合写像Ueo、データの統合写像Ud、プロセスの統合写像Up、アクターの統合写像Uac、データストアの統合写像Ustがある。

```
[("thesis",["tname"])]:Ue
[(("ST_Student","t1","thesis"),"handin")]:Ueo
[("ListofThesis","list"),("Thesis","tname")]:Ud
[("Store",Office.update)]:Up
[("student",([Student.tname],
  [Student.write,Student.finish]))]:Uac
[("Storage",([Office.counter,
              Office.list],[]))]:Ust
```

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。

図2 仕様

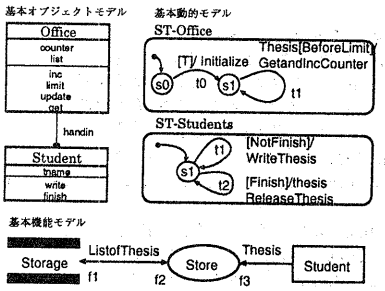

図3 例題の基本モデルの図的表現

3. 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既に提案している不変表明を用い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。不変表明を用いた一貫性検証は、述語論理を用いた定義や証明が必要であり、定理証明系HOLを用いることにより支援

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

図4　識別子の意味記述

する。

3.1 HOLの概要

HOLは、理論のモジュール性、ユーザ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。

- 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。
- 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。

3.2 不変表明を用いた一貫性検証

3.2.1 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。

統合写像が与えられた動的モデルでは、アクション

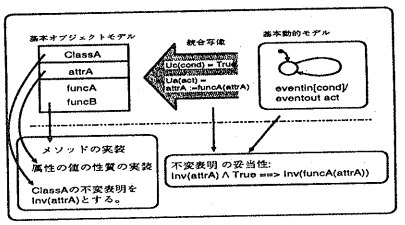

図5　不変表明を用いた一貫性検証

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検証して、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。ここで、`T`は真を意味する。

遷移`t`に対して、

- 遷移状態前が初期状態の場合

  $T \Rightarrow Inv(c)$

- 遷移`t`が初期状態からの遷移でない場合

  $Inv(c) \wedge bexp \Rightarrow Inv(c)$

ここでは、遷移条件論理式を`bexp`とする。

次に、例題に対し不変表明を用いた一貫性検証を適用する。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。

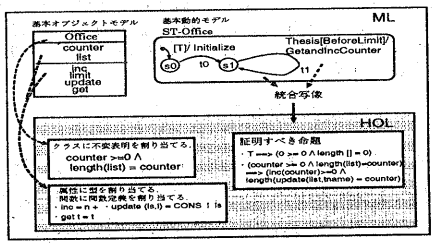

図6　不変表明を用いた一貫性検証の例題への適用

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みたしているか、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みた

しているかを検証する。

この検証法では識別子の意味記述を基に、基本オブジェクトモデルに出現する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。ここでは、クラスClassが持つ属性attrをClass.attr、クラスClassが持つ関数funcをClass.funcという形式で表現する。また、HOLの項は(--‘ ‘--)を用いて定義し、型は(==‘: ‘==)を用いる。

- 属性識別子に対する割り当て
  - Office.counterに対して、
    (==‘:num‘==)型の(--‘counter‘--)を割り当てる。
  - Office.listに対して、
    (==‘:string list‘==)型の(--‘list‘--)を割り当てる。
  - 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
  - Office.incに対して、
    (--‘inc(n:num)=n+1‘--)を割り当てる。
  - Office.limitに対して、
    (--‘limit(day:num)= day<pre_defined_limit‘--)を割り当てる。
  - Office.updateに対して、
    (--‘update(ls:string list,s)=s::ls‘--)を割り当てる。
  - 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不変表明を割り当てる。
  - Officeに対して(--‘0 <= counter /\
    length (list) = counter‘--)を割り当てる。

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が存在するので以下の命題を説明すればよい。

- 状態遷移t0
  (--‘T ==> 0 <= 0 /\ (LENGTH[] = 0)‘--)
- 状態遷移t1
```
(--‘0 <= counter /\
(LENGTH list = counter) /\ limit
==>
0 <= inc counter /\
(LENGTH (update (list,tname)) =
inc counter)‘--)
```

3.2.2 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。モデルの属性識別子や関数識別子に割り当てた各々の概念をHOL上に宣言する。

新しい理論の名前をinvとして、宣言する。

親のtheoryとして、HOLの組み込みのtheory stringを宣言する。これにより、theory stringを用いることが可能となる。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listが含まれている。

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

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

属性識別子Office.counter、Office.list、Student.tnameに対して割り当てた項定数を以下のように宣言する。

```
- 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
```

関数識別子Office.inc、Office.updateに割り当てた関数定義を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,l) = CONS l ls‘--));
val it =
|- !ls l. update (ls,l) = CONS l ls : thm
```

関数識別子Office.limitとOffice.getに割り当て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、予め外部の環境を作成する。外部の環境の詳細を予め用意することにより、その環境を参照する関数の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の定義を行う。

Epre_defined_limitは締切日を意味し、ここでは10として宣言する。Edayは今日の日付を意味し、ここでは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。

Office.limitとOffice.getに対して割り当て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。

```
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:
```

証明すべき命題を、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 [] = 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
```

prop_t0は、以下の手順で証明可能である。

```
- val theorem_t0 = DISCH(--`T`--)(CONJUNCT2(CONJ
(ASSUME (--`T`--)) (CONJ (EQT_ELIM
(ARITH_CONV(--`0 <= 0`--)))(CONJUNCT1 LENGTH))));
val theorem_t0 =
|- T ==> 0 <= 0 /\ (LENGTH [] = 0):thm
```

同様に、prop_t1に関しても証明可能である。

4. 検証フレームワーク

前章では、不変表明を用いた一貫性検証に対してHOLを用いて検証環境を作成し、検証を行った。し

かし、形式的モデルを用いた検証法は、これらの他にも無数に存在する。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した部分な部分であり予め実装しておく。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

検証アプリケーションでは、MLの参照型を用いて保持されているモデル情報に対して加工/選択を行い、対応するHOL上の概念に翻訳して検証環境を構築することになる。

加工/選択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。また、HOL上の概念に翻訳を行うためには、各々の識別子に対してHOL上で取り扱われる概念を用いて表現し、また、構築されたモデルの構成要素に対してHOL上の環境の基で解釈を行う必要がある。したがって、検証フレームワークでは以下のもの(図7)が定義される。

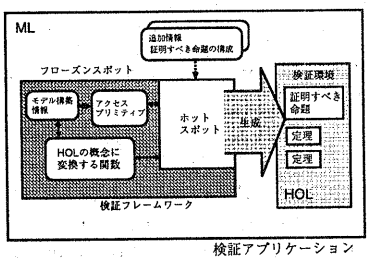

図7 検証フレームワーク

- モデル構築情報を格納する参照型変数。
- モデル構築情報に対してアクセスするプリミティブ。
- HOLの概念に変換する3つの関数
  - 割り当て関数
    モデルの構成要素の意味をHOL上で取り扱われる概念に割り当てる。
  - 翻訳関数
    モデルで定義されている式に対し、HOL上の環境のもとにおける解釈を与える。

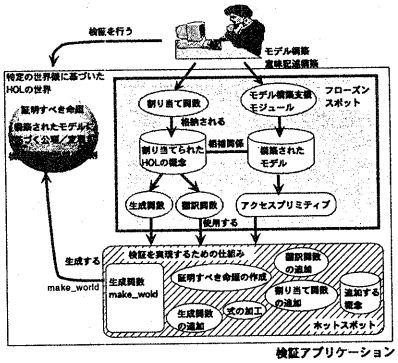

図8 検証フレームワークのアーキテクチャ

  - 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。

検証フレームワークのアーキテクチャ(図8)は、割り当て関数により構築されたモデルにHOL上で取り扱われる概念を割り当て、翻訳関数と生成関数によってHOL上の世界を生成するものである。検証フレームワークのホットスポットでは、検証環境を生成する生成関数make_worldを作成する。make_worldでは、フレームワークに実装されている割り当て関数、翻訳関数、生成関数、アクセスするプリミティブを用いて作成する。そして、作成したmake_worldを実行することにより、容易に検証環境を得ることができる。

4.2 割り当て関数、翻訳関数、生成関数

例題の割り当て関数、翻訳関数、生成関数を以下に示す。MLでは、クラスClassが持つ属性attrをAO_CLASS("Class","attr")の形式で表現し、型はAOである。また、クラスClassが持つ関数funcをFO_CLASS("Class","func")の形式で表現し、型はFOである。

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。

```
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_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。

翻訳関数に関しては、例題ではアクション式`AExp`、論理式`BExp`に対するものがある。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`が定義されている。

アクション式に関する翻訳関数`tl_aexp`は`AExp -> {redex:term,residue:term} list`型であり、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)
```

このアクション式をそれぞれ`AEXP1`、`AEXP2`と置くと、翻訳関数`tl_aexp`を適用した結果は以下となる。

```
- tl_aexp (AEXP1)
val it = [{redex=``counter``,residue=``0``},
{redex=``list``,residue=``[]``}]
  : {redex:term, residue:term} list

- tl_aexp (AEXP2)
val it =
[{redex=``counter``,residue=``inc counter``},
{redex=``Ecurrentpaper``,residue=``get tname``},
{redex=``list``,residue=``update (list,tname)``}]
  : {redex:term, residue:term} list
```

例題の論理式`Office.limit ()`に翻訳関数`tl_bexp`を適用すると以下になる。ここでは、`Office.limit ()`を`BEXP`と置く。

```
- tl_bexp (BEXP)
val it = ``limit`` : 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

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、容易に検証環境を得ることができる。ホットスポットとして、クラスに対して不変表明を割り当てる関数`bind_inv`を追加し、クラスを与えると証明すべき命題を返す翻訳関数`tl_inv`を追加する。

クラスに対する不変表明は割り当て関数`bind_inv`を用いて割り当てる。

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

追加する翻訳関数は`"Office"`を引数にとり、割り当て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返す関数である。

```
- tl_inv "Office";
val it =
  [``T ==> 0 >= 0/\(LENGTH [] = 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、その全ての証明すべき命題を変数`prop`に割り当てる。

5. まとめ

本研究では、既に提案している形式的モデルをML上に実装をおこなった。不変表明を用いた一貫性検証に対し、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、検証支援環境を検証フレームワークの概念を用いて整理した。

本研究では、3つの検証法を例題に適用したにとどまっている。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、その評価を行うことが挙げられる。

参考文献

1) Ramgaugh,J.Blaha,M.,Premerlani,M.,Eddy,F. and Lorensen,W.: Obect-Oriented modeling and design, Prentice-Hall International, 1991.
2) 青木利晃, 片山卓也: オブジェクト指向方法論のための形式的モデル, , 日本ソフトウェア科学会学会誌 コンピュータソフトウェア.
3) Toshiaki Aoki and Takuya Katayama: Formal Model Approach for Reliable Object-Oriented Information System Design, ,Sci/ISAS '98,pp.228-235,1998.
4) Toshiaki Aoki and Takuya Katayama: Unitication and Consistency Verification of Object-Oriented Analysis Models, ASPEC '98,1998.
5) M.J.C.Gordon, T.F.Melham: Introduction to HOL, CAMBRIDGE UNIVERSITY PRESS, 1993