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

Title	オブジェクト指向方法論のための形式的モデルの検証
Author(s)	石田，至
Citation	
Issue Date	1998-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1125
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形式的モデルの検証

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

石田至

1998 年 2 月 13 日

Copyright © 1998 by Itaru Ishida

要 旨

本稿では、オブジェクト指向方法論のための形式的モデルに対して、そのモデルに関する検証支援機能を有した検証フレームワークを構築する。

目 次

1 始めに **1**

1.1 背景 1

1.2 目的 2

1.3 論文構成 2

2 FO∀M の導入 **3**

2.1 オブジェクト指向開発 3

2.2 FO∀M の概要 4

3 検証フレームワークの設計 **6**

3.1 フレームワークの導入 6

3.2 検証フレームワークの機能 7

3.3 FO∀M の構造の分析 8

3.4 フレームワークの設計と構築の方針 9

4 本研究で用いる検証系について **11**

4.1 フレームワークの実装に用いる検証系の選択 11

4.2 HOL 12

5 検証フレームワークの実装 **15**

5.1 FO∀M を構成する理論の実装 15

5.2 FO∀M の各モデルの実装 16

5.2.1 識別子の実装 17

5.2.2 式/写像の実装 23

5.2.3 規則/制約等の実装と検証の補助のための定義 32
5.3 モデル情報 35
5.4 検証の方針 35
5.5 実装のまとめ 36

6 検証フレームワークを用いた検証例 37
6.1 例題 37
6.1.1 基本オブジェクトモデル 37
6.1.2 基本動的モデル 39
6.1.3 基本機能モデル 42
6.1.4 統合モデル 42
6.1.5 検証例 44

7 考察 49
7.1 検証フレームワークとして要求された機能に関する考察 49
7.2 検証フレームワークの利用に関する考察 51

8 まとめ 52
8.1 まとめ 52
8.2 今後の課題 52

図 目 次

2.1 FO∀M の概念図 5

3.1 FO∀M の検証フレームワークの構築の方針 10
3.2 FO∀M の各モデルの構築 10

5.1 FO∀M 理論の構築の方針 16
5.2 文字列から識別子への写像 19
5.3 識別子間の関係 19
5.4 識別子の CONVERSION 24
5.5 継承式を構成する要素からの式への写像 28
5.6 継承式の写像を構成する要素からの継承式の写像型への写像 32
5.7 FO∀M で記述されたモデルの検証方針 36

6.1 OMT の記法で記述したオブジェクトモデル 38
6.2 OMT の記法で記述した動的モデル 39
6.3 OMT の記法で記述した機能モデル 42

第 1 章

始めに

1.1 背景

近年のシステム開発において、オブジェクト指向開発が注目されている。多くのオブジェクト指向開発のための方法論が提案されており、実際の開発に用いられるようになってきている。また、大規模なシステムを対象とした開発には計算機の支援が必要である。しかし、従来のオブジェクト指向方法論では形式的な取り扱いが十分でなく、計算機による支援を困難なものにしている。また、開発の分析段階における対象システムのモデルの性質に関して検証をおこないたいという要求もある。作成した対象システムのモデルの性質の検証には、計算機による支援が必要となる。

そのような目的から、オブジェクト指向開発を形式的にすすめるために、従来の経験的蓄積を用いて構成されているオブジェクト指向開発法をもとにして、集合と関数の概念をもとに形式化をおこなった、青木の FO∀M (Formal model for Object-oriented Analysis Model)[1] がある。

FO∀M を用いて対象システムを分析し、モデル化することで、オブジェクト指向開発を形式的にすすめることが可能となる。また、計算機による支援も容易におこなうことが可能となる。

1.2 目的

FO∀M を用いて対象システムを分析することで、オブジェクト指向開発の分析段階で構築するモデルを形式的に扱うことが可能になる。さらに、対象システムの分析に関するさまざまな計算機による支援が可能になると考えられる。

支援環境に対する、対象システムのモデルに関して、モデル化の一貫性の検証、対象システムの性質を理解するためのモデルに対するさまざまな性質の検証等が挙げられる。また、FO∀M は用いられている定義が多くその論理的構造が複雑であるため、その取り扱いの上でも計算機の支援は重要であると考えられる。

以上のような要求を満たすために、計算機上に FO∀M の理論の公理系を実装する。そして、FO∀M で記述されたモデルの検証をおこなうことが可能な検証フレームワークを構築し、その有効性を確認する。

1.3 論文構成

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 FO∀M に関する簡単な概要を説明する。そして 3 章で本研究で目的とする検証フレームワーク構築に関する分析をおこない、4 章で本研究で用いる検証系 HOL の紹介をする。5 章で検証フレームワークの実装に関して説明し、6 章で実例を用いて検証フレームワークでの検証の例を示す。そして、7 章で考察をおこない、 8 章でまとめと今後の課題を述べる。

第 2 章

FO∀M の導入

この章では本研究に用いる FO∀M について、その背景としている概念と理論を解説する。

2.1 オブジェクト指向開発

オブジェクト指向開発は、システム開発において従来用いられてきた機能を中心に分析をおこなう構造的開発法とは異なり、オブジェクトを中心として分析をおこなう開発である。

システムに存在するオブジェクトを識別し、そして、システムの各機能はオブジェクト間の通信によって実現する。このような概念をもとに、システムの分析をおこない、設計、実装をおこなうのがオブジェクト指向開発である。

最近では、オブジェクト指向開発が注目されており、、実際のシステム開発に用いられる事例も増えてきている。

しかし、従来のオブジェクト指向方法論では対象システムのモデルの形式的とり扱いが十分になされておらず、計算機の支援を妨げる要因となっている。

オブジェクト指向開発を 進めるための、多くのオブジェクト指向方法論が提案されている。

その中でも、OMT [4] は、システムを直交する 3 つの側面 (構造的側面、動作的側面、機能的側面) においてモデル化をおこなう分析する手法を提案している。これらの 3 つの側面はシステムの性質に関して、それぞれの側面における強力な分析法を提供する。

しかし、OMT においてもモデル化に関しての形式的取り扱いが十分ではなく、計算機による支援を十分におこなうことができていない。

2.2 FO∀M の概要

大規模なシステム開発においては、計算機の支援が不可欠であるが、従来のオブジェクト指向方法論では前述した理由から計算機による支援が困難である。

このような要求に対して、オブジェクト指向方法論のための形式的モデルである FO∀M (Formal model for Object oriented Analysis Model) が青木によって提案されている。FO∀M を用いて対象システムをモデル化することで、オブジェクト指向開発の分析段階において、形式的な分析モデルを構築することができる。

その形式化の方針は、OMT で用いられている、システムの直交する 3 つの側面における分析モデルをもとにしている。

3 つの側面はシステムの構造的側面、動作的側面、機能的側面であり、それぞれシステムの性質の主成分ととらえることができる。

この 3 つのモデルはシステムの持つ性質の直交した主成分を反映するものであり、それぞれの側面において独立した分析モデルを提供している。

FO∀M では、これらの視点から分析した各モデルを基本モデルとしている。

また、各基本モデルをもとにして、対象システムに関する一貫した分析モデルを定義するための統合写像のメカニズムを導入した統合モデルを提供している。

1. 基本モデル
   各基本モデルでは、モデルの基本集合として識別子を用いる。識別子はそれぞれの側面を構成する要素の最小単位であり、それぞれの側面に固有な概念を抽象化したものである。識別子の意味記述 に関しては別途ドキュメント化する。この識別子を基本集合としてモデル化をおこなうことにより、システムを 3 つの側面に独立に分解して定義できる。要求仕様に記述されている詳細な機能などの情報は、それぞれの側面の識別子に対する意味記述として整理される。

2. 統合モデル
   独立に定義された基本モデルでは、側面は直交しているが、同じ対象システムを射影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

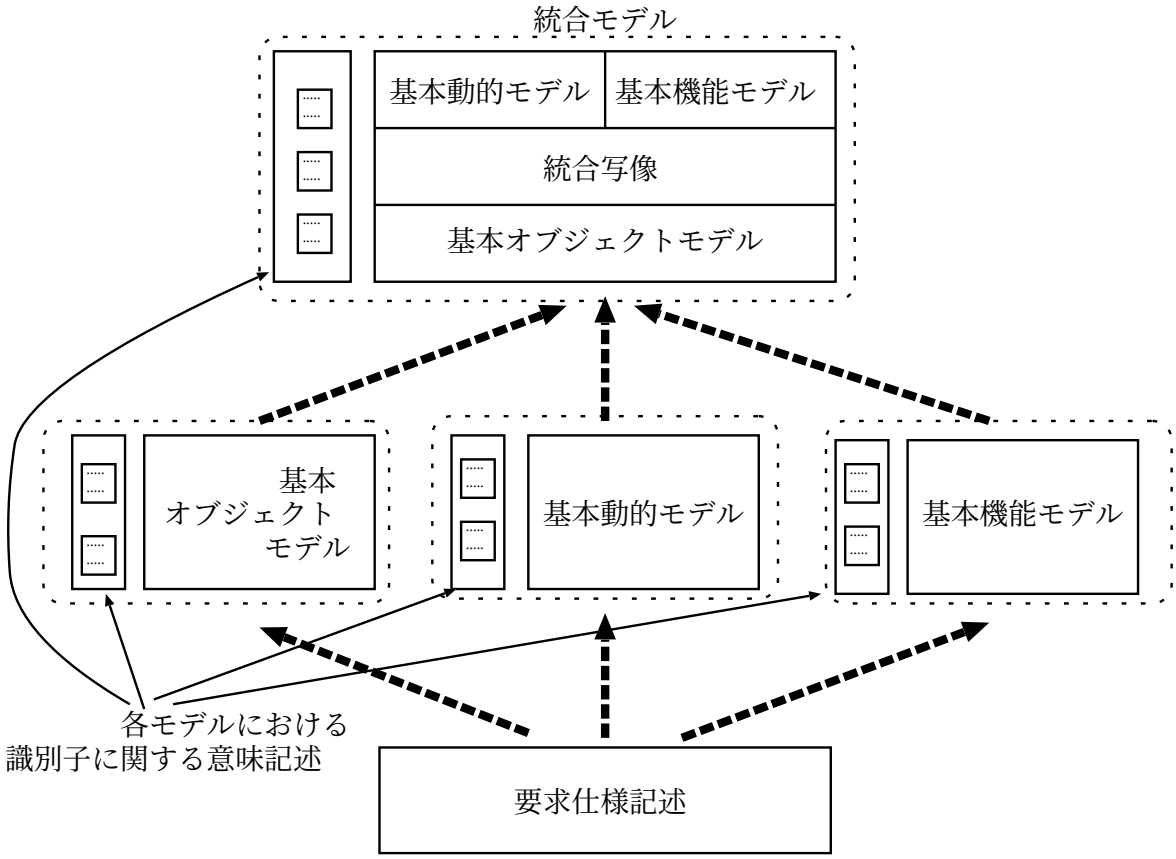

図 2.1: FO∀M の概念図

こで、意味が共通する部分を対応づけるメカニズムである統合写像の概念を導入する。このような対応関係を導入し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 つに統合されたモデル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もとにおこなわれ、対応関係を示す統合写像を形式的に定義する。これにより基本でモデルでは意味記述として比形式的に記述されていた部分が段階的に形式的記述に変換されることになる。

このように、統合モデルでは独立に構築され分析されたモデルとそれらの対応関係が定義されており、1 つの一貫した分析モデルを定義している。

第 3 章

検証フレームワークの設計

この章では、FO∀M の検証フレームワークを構築する際に考慮するべき事を洗い出す。そして、実際に検証フレームワークを構築するための指針を決定する。

3.1 フレームワークの導入

フレームワークはシステム開発において、ある特定の領域に対する再利用可能な枠組を提供する。フレームワークを利用して対象システムを構築する場合、フレームワークで提供されている枠組に、構築するシステム固有の情報を組込むことで全体を構築できる。フレームワークを用いる利点は、同様の領域のシステムを構築する場合に、その再利用性が高いことである。

ここで、FO∀M を用いた開発について考える。FO∀M はオブジェクト指向開発のために、対象システムを形式的に記述するためのモデルとその理論を提供している。FO∀M を開発に利用する場合、対象システムの情報をもとに FO∀M の枠組を利用して形式的にモデル化をおこない、開発を進める。

よって、FO∀M で記述されたモデルに関して、そのモデルに関する検証をおこなうことを考慮した場合、FO∀M の理論をフレームワークとして構築することが有効であると考えられる。FO∀M の理論のフレームワークを構築しておけば、FO∀M を用いて構築されたモデルに関して、一貫した検証の枠組を提供することが可能となる。

3.2 検証フレームワークの機能

FO∀M に関する検証フレームワークを考えた場合、以下のような機能を持つべきであると考えられる。

- FO∀M で構築されたモデルの構文チェック
  FO∀M を用いて構築されたモデルは、その構文が FO∀M で定義されている構文にしたがっている必要がある。そのため、構築したモデルに関してその構文を簡単にチェックできる機能が必要である。

- FO∀M で構築されたモデルの一貫性の検証
  FO∀M を用いて作成されたモデルに関して、そのモデルの性質が FO∀M の理論との一貫性を保持しているかを検証する。よって、モデルに関する意味的な一貫性検証の機能が必要である。

- 対象システムの性質の検証とその理解の支援機能
  対象システムのモデルの性質に関して、それがどの規則からどのように示されるかを検査したい要求がある。モデルの性質を検証しその証明過程をたどることで、その性質に関する理解を深めることができる。よって、モデルの検証に関して、その証明系列に関する健全性を保持する機能が必要である。また、システム開発において対象システムが満たしている性質が明らかであることは少ない。モデル構築の後にそのモデルに対して性質を検証していくことで、そのシステムの性質が明らかにできる。検証フレームワークを利用することで、対象システムの性質の理解が進み、その性質が明らかにできるような機能が必要である。

以上のような機能を持つ検証フレームワークを最初から構築するのはかなり困難である。

そこで、既存の検証系を利用することを考える。検証系を利用した場合、FO∀M の理論を公理系として構築することができる。その場合、FO∀M の理論で定義されているさまざまな定義を公理群として定義することが可能であり、また、対象システムの性質の検証を、証明によっておこなうことが可能となる。性質の検証系を証明によっておこなうことは、その証明過程を理解することで対象システムの性質の深い理解につながる。また、FO∀M の識別子の意味記述に関しても、証明をおこなう検証者が考慮することができる。

検証系上にフレームワークを実装することで、FO∀M の理論を公理系として実装できる。実装した公理系を利用すればFO∀M に関する性質を証明によって検証できる。また、検証系を用いることで、証明の健全性を保証できる。

既存の検証系の上に FO∀M を扱うことが可能な公理系を実装する。そして、その公理系を利用して、対象システムの情報から性質に関する証明をおこなうことが可能なFO∀M の検証フレームワークを構築する。

3.3 FO∀M の構造の分析

FO∀M の検証フレームワークを構築するにあたり、FO∀M の理論をどのように検証系上に実装するかを考慮する必要がある。そのために、実装することを考慮してFO∀M の理論の論理的構造を明らかにする。そして、その構造を考慮した上での FO∀M の理論の実装方針を検討する。

FO∀M の構造

FO∀M の理論は、以下のような論理的構造を持つと考えることができる。

- FO∀M の理論を構成するモデルに関する構造
  FO∀M の理論は複数のモデルから構成されている。各基本モデルはそれぞれ他の基本モデルの定義から独立に定義されている。統合モデルは各基本モデルの定義をもとに、統合写像の概念を導入して定義されている。構造としては、基本モデル間の独立性が高く、統合モデルとの間には依存関係が存在する。

- FO∀M の各モデルに関する構造
  FO∀M を構成する各モデルは、それぞれ識別子や式などの各モデルにおける対象システムの構造を表現する要素と、その要素が満たすべき性質の定義から構成される。それらの定義は、基本的に各モデルの中で閉じている。

このような構造を踏まえて、FO∀M の検証フレームワークの構築の方針を決定する。

3.4 フレームワークの設計と構築の方針

以上の分析から、FO∀M の検証フレームワークの構築に関する設計と、構築の方針を決定する。FO∀M の検証フレームワークは、FO∀M の理論を各基本モデルに関してそれぞれ独立した公理系として実装する。統合モデルはその基本モデルの公理系をもとに定義される公理系として定義する。

各モデルのおのおのの公理系に関しては、そのモデルを構成する要素である識別子や式等をデータ構造と見て定義し、その識別子や式などの上に定義される性質を、データ構造の性質 に関する定義として定理群を構築する。

このように、各モデルの公理系と FO∀M におけるデータ構造と基本的性質の定義を核として、その上で証明される定理からなる検証フレームワークを構築する。

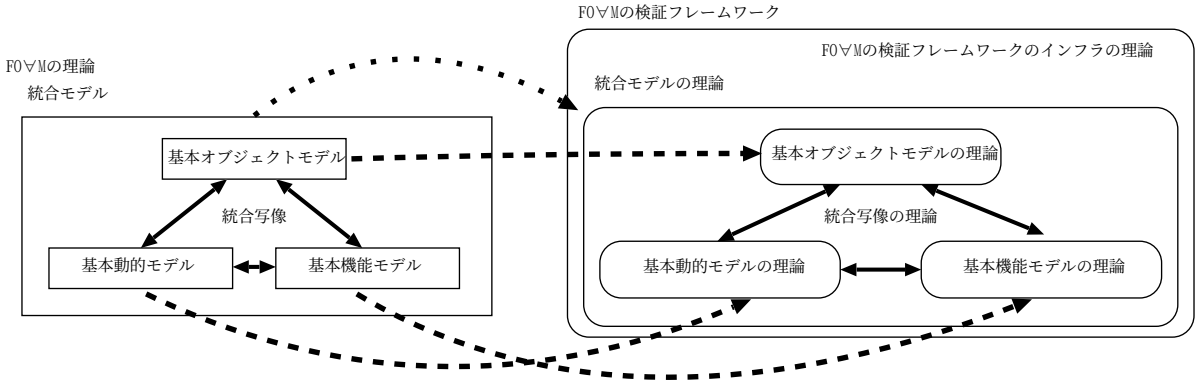

図 3.1: FO∀M の検証フレームワークの構築の方針

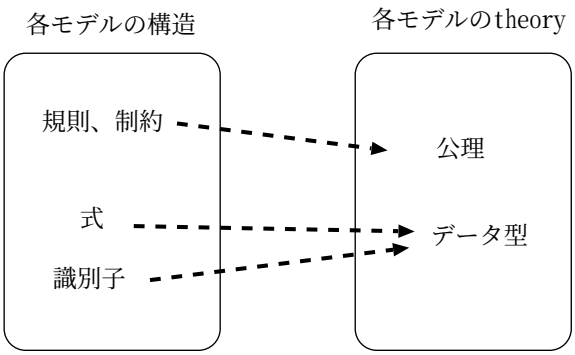

図 3.2: FO∀M の各モデルの構築

第 4 章

本研究で用いる検証系について

本章では検証フレームワークを構築する際にベースとする検証系である HOL に関して、その機能と実装の際の利点を解説する。

4.1 フレームワークの実装に用いる検証系の選択

検証フレームワークの構築には検証系を用いるが、現在利用できる検証系にはさまざまな種類がある。そこで、検証フレームワークのベースとして利用する検証系を決定しなければいけない。

FO∀M の理論を実装するための検証系として満たすべき性質を挙げると、FO∀M で用いられているさまざまなデータ型の定義が可能で、証明に関する多くの定義が可能であることが必要であると考えられる。

広く利用されている検証系で、高階論理をもとにしたユーザ定義型を利用できる検証系に PVS がある。PVS は検証システムと検証の対象を記述するための仕様記述言語がセットになった検証系である。PVS は環境として完結している検証系である。そのため検証環境を構築するベースと見た場合に、PVS の環境の上にさらに検証フレームワークの環境を構築することは困難であった。また、PVS では複雑な型を構成する際に問題があった。このような理由から、目的とする検証系としては、不十分である。

FO∀M の性質を考慮し、本研究では実装する検証フレームワークのベースとする検証系として、後述する理由から HOL[5] を選択した。

4.2 HOL

ML 上に実装された検証系で、以下の特徴を持つ。

- 高階論理をサポートしている。
  高階論理をサポートしていることにより、ラムダ計算を利用したユーザによる柔軟な型の定義が可能となっている。

- 公理系のモジュール的な扱いが可能である。
  HOL では 1 つの公理系に関する公理や定理を、theory という単位で扱うことが可能となっている。公理系は theory 単位でまとめて扱うことができ、他の公理系の定義と明確に分割することができる。また、公理系間の依存関係も定義できる。証明は theory を呼び出すことによって、そこで定義されている公理系に関する性質を利用しておこなわれる。また、組込みの theory が豊富で文字列やリストに関する基本的な性質に関する公理系に関しては十分検証可能である。

- ML 上に HOL の検証環境を補助する関数が定義されている。
  検証系が ML 上に構築されていることから、ML 環境側から HOL の項や定理を扱う関数が提供されている。

  - 検証系に関する基本的な定義のための ML 関数が提供されている。
    ユーザが型を定義する際に、その型に関する定義を自動的に証明する ML 関数や、ある種の定型の証明に関して自動的にその証明をおこなうための ML 関数が豊富に提供されている。
  - 手続き的な定義や証明が可能。
    証明の過程を ML 変数に保存することが可能で、後にその変数を利用して証明や定義をおこなうことが可能になっている。

- HOL 上で証明を進めるための重要な仕組みが ML 関数として提供されている。
  HOL 上の証明において、以下の重要な要素が ML 関数の形で提供されている。これらもまた、手続き的利用が可能である。

  - RULE
    公理や定理をその恒真性を保存したまま、推論規則により変形する操作を提供する。

- CONVERSION
  ある項に対して、恒真性を保ったままその項と、別の項を含む等式の形の定理に変形する操作をおこなう。証明において単純な書き換えでは求められない変換や、単純な変形では証明できない項と項の間の等価性を示す際に用いる。
- TACTIC/TACTICAL
  ゴールを設定しておこなう証明のに、ゴールをその満たすべき性質を保存したままサブゴールに分割する操作をおこなう仕組みを TACTIC として提供している。TACTICAL は TACTIC を組み合わせる仕組みである。TACTIC と TACTICAL を組み合わせることで、証明を手続き的な操作でおこなうことができる。また、その組み合わせを新たな TACTIC として定義しておくことで、同様の証明に用いることが可能になる。TACTIC や TACTICAL によって定型の証明に関して証明系列を定義でき、戦略的な証明が可能となる。

HOL を FO∀M の検証フレームワークを構築するベースとして見たときに、theory のモジュール性や、柔軟にユーザ定義型を作ることがが可能なこと、そして、ML 環境を用いた検証補助環境を構築できる点で有効であると考えられる。

HOL の記法の説明

本論文では HOL における論理式の表現がいくつか示される。そのため、HOL における論理式の表現の意味と、それに対応する論理式の表現を次の表に示す。

HOL の項や定理は ML 上の抽象データ型で表現され、(--‘ ‘--) で囲まれた項で表される。また、⊢ が使われた論理式は、公理かまたは証明された定理を示す。その論理式では、⊢ の左側の部分が仮定を表し、右側の部分が結論を示す。

HOL での記法	論理式の表現	意味
`T`	$\top$	真
`F`	$\bot$	偽
`~`t	$\neg t$	t の否定
t_1`\/`t_2	$t_1 \vee t_2$	t_1 または t_2
t_1`/\`t_2	$t_1 \wedge t_2$	t_1 かつ t_2
t_1`==>`t_2	$t_1 \Rightarrow t_2$	t_1 ならば t_2
t_1`=` t_2	$t_1 = t_2$	t_1 と t_2 は等しい
`!`x`.`t	$\forall x.\ t$	t における全ての x
`?`x`.`t	$\exists x.\ t$	t に x が存在する
`?!x.t`	$\exists_1 x.\ t$	t において x が一意に存在する
`@`x`.`t	$\epsilon x.\ t$	t における (条件に合う) そのような x
`\`x`.`t	$\lambda x.\ t$	x に関するラムダ計算 t
`(`t`=>`t_1`|`t_2 `)`	$(t \rightarrow t_1, t_2)$	もし t ならば t_1、そうでなければ t_2

第 5 章

検証フレームワークの実装

本章では、FO∀M の理論をどのように実装したかを、代表的な実装例について実例を示し解説する。

5.1 FO∀M を構成する理論の実装

FO∀M は 3 つの基本モデルと、基本モデルをもとにそれぞれを対応づけた統合モデルから構成される。各基本モデルはそれぞれ独立していて、他の基本モデルに影響を与えるべきではない。そこで、FO∀M の理論の実装に HOL の theory のモジュールとしての性質を利用する。各基本モデルを独立の theory として定義することで、１つの基本モデルの中で閉じている性質に関しての検証は、関連するモデルの theory だけを利用して検証をおこなうことができる。そして、統合モデルの理論は、theory の依存関係を利用して各基本モデルの theory の上に成立する theory として定義する。統合モデル上での検証は、各基本モデルの検証に加えて、統合モデルの性質に関する性質を検証できる。

このように、FO∀M を構成するの各モデルの理論を HOL 上の theory として実装した。検証フレームワークは FO∀M の理論を構成する各モデルの theory から構築される。

次節からは、FO∀M の各モデルの理論を構成する要素を代表的な例を用いて、HOL 上にどのように実装したか解説する。

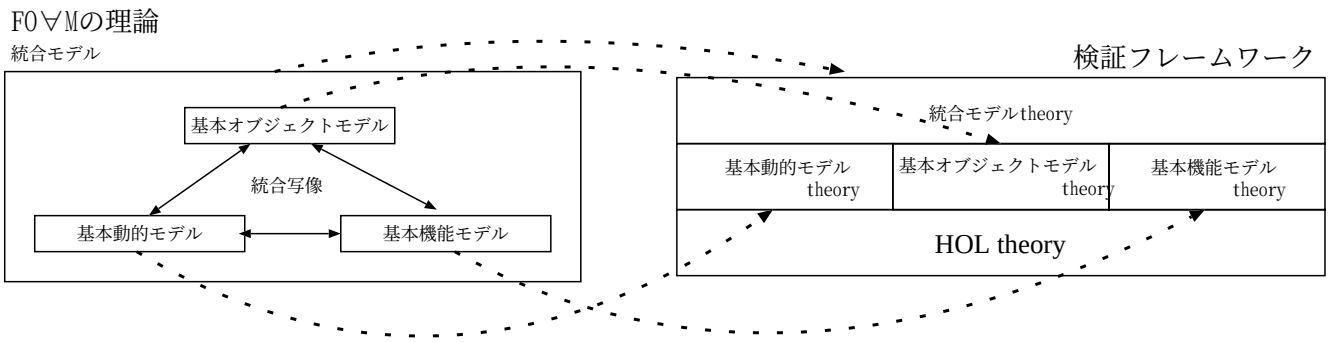

図 5.1: FO∀M 理論の構築の方針

5.2 FO∀M の各モデルの実装

FO∀M の理論全体の論理的構造を考慮して、各モデルを HOL の独立の theory として定義するよう決定した。次に各モデルをどのように theory として定義するかを決定しなければならない。

そのために、FO∀M の各モデルの理論を構築している要素を分析し、検証フレームワークに実装しやすいように再構成する必要がある。そこで、そのような視点から FO∀M の各モデルの構造を分析すると、FO∀M の各モデルを構成する要素を大きく以下の 3 つに分類することができる。

- 識別子
  FO∀M の理論の最小構成要素。
- 式や写像
  識別子から構成されるモデルの構造を表現する要素。
- 制約/規則など
  識別子や式などの上に定義される性質。

この分類をもとにして、検証フレームワークにおける FO∀M の各モデルの構築をおこなう。

5.2.1 識別子の実装

識別子は、FO∀M 理論を構成する最小単位の要素である。各基本モデルは、識別子を基本集合として定義される。よって、各基本モデルの構成要素は、最終的には識別子のレベルまで分解することができる。各識別子はそれぞれ他の識別子と明確に区別されるものなので、それぞれの識別子を HOL 上の 1 つの型として実装することにする。識別子を表現する型を識別子型とする。また、識別子は FO∀M を構成する最小要素であるため、検証のために識別子同士を比較することが必要になる。詳しくは、識別子型の CONVERSION の節で述べるが、比較をおこなうためには、識別子型に何らかの値を持たせる必要がある。よって、識別子型の値として文字列型を持つことにする。結果として、識別子型は、文字列型から識別子型への関数を用いて文字列型から構成される型として実装することになる。識別子型を構成する関数は、識別子型の型コンストラクタととらえることができる。

識別子型の定義は、HOL の型を定義するために用意されている HOL の組込みの ML 関数 define_type を利用して以下のようにおこなう。

クラス識別子を例にした識別子型の定義

```
val ClassID_Axiom =
          define_type{name="ClassID_Axiom",
                      type_spec=`ClassID = CLASSID of string`,
                      fixities = [Prefix]};
```

HOL では、型を定義するための関数がいくつか用意されている。

この関数は HOL でユーザデータ型を定義するときに用いる関数で、型を構成するための識別子と現在 HOL 上で利用できる型から新しい HOL の型を作るための公理を作る。

上のように define_type を実行すると、ClassID という型とその型を構成する要素に関する ClassID_Axiom という名前の公理が自動的に証明される。そして、その公理を ClassID_Axiom という名前の ML 変数に保存している。結果として、ClassID 型と、文字列型から ClassID 型を構成する Prefix のラムダ関数 CLASSID が定義される。

$$\vdash \forall f.\ \exists_1 fn.\ \forall s.\ fn(\mathrm{CLASSID}\ s)\ =\ f\ s$$

型の構成という視点からとらえると、CLASSID は文字列型からクラス識別子型を構成する型コンストラクタである。

このようにして作られた型は HOL 上では、以下のように表現される。

```
(--` CLASSID "classname" `--)
```

これは HOL における、値として文字列型の classname を持つクラス識別子型の項である。

また、define_type で定義された型に関しては、HOL に組込みの ML 関数 prove_constructors_one_one でその型に関する一意性の定理を自動的に証明することができる。

```
val ClassID_11 =
    save_thm ("ClassID_11", prove_constructors_one_one ClassID_Axiom);
```

そのように自動的に生成した、クラス識別子の一意性に関する定理が以下の定理である。これはクラス識別子に関して、「2 つのクラス識別子が等しいことは、 クラス識別子を構成している文字列が等しい」ことと等しいことを示す定理である。この定理は ML 変数 ClassID_11 に保存される。

$$\vdash \forall\, \mathrm{s}\ \mathrm{s}'\ .\ (\mathrm{CLASSID}\ \mathrm{s} = \mathrm{CLASSID}\ \mathrm{s}') = (\mathrm{s} = \mathrm{s}')$$

このように、クラス識別子型は文字列型の集合の要素 から、クラス識別子の型コンストラクタにより、対応するクラス識別子の要素へと 1 対 1 に写像されると考えることができる。

以上の定義でクラス識別子を例に識別子を表現する型の構成を示した。他の識別子に関しても、型コンストラクタが異なるだけで基本的に同様の定義で識別子型を構成する。

各識別子型は、それぞれの識別子を構成するための型コンストラクタが異なるので、同じ文字列型から構成されてもそれぞれ明確に区別される。

識別子型の等価性

HOL の証明は、主に公理や定理を利用した項の書き換えによって行われる。書き換えには、等式の形の公理や定理が用いられる。そして、新たに証明した定理も書き換え規則

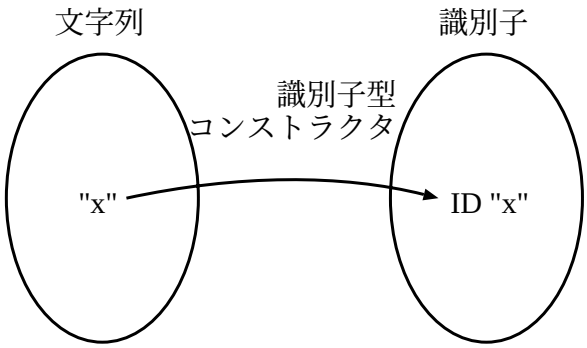

図 5.2: 文字列から識別子への写像

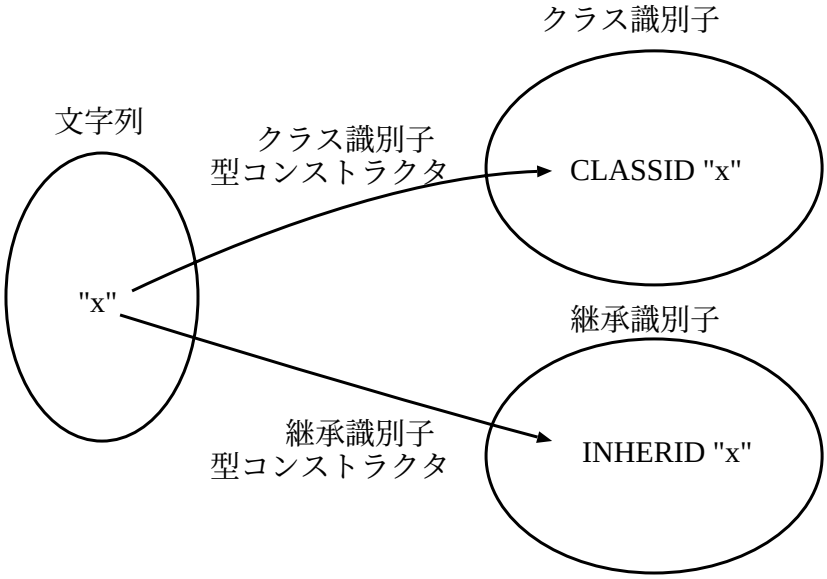

図 5.3: 識別子間の関係

として利用することができる。そして、HOL の等価性の証明に関しては、通常以下の書き換え規則を利用して行われる。

$$\vdash (x = x) = T$$

この書き換え規則は、書き換え対象の項を見て、項の左辺と右辺が等しければ真に書き換えるという、値の考慮を全く行わない書き換えである。よって、書き換えでは項の形が異なるもの同士の比較は行えない。そこで、HOL では複雑な等価性に関する証明の道具として CONVERSION という仕組みを用意している。

CONVERSION は単純な書き換えでは示すことができない項の等価性に関する定理を示すための機能である。

FO∀M では、システムの構成要素に関して等価性の比較を行う際には、最終的にその最小構成要素である識別子の段階で比較を行うことになる。

上の書き換え規則では、そこで述べたように等しいもの同士の比較しか行うことができない。よって、識別子の段階で等価性を示すことが可能な CONVERSION を用意する必要がある。

その準備として、識別子型の定義を文字列型からの写像として定義してある。この性質を利用することで、識別子の比較に関する定義が可能になる。以下に、クラス識別子を例として識別子の CONVERSION の定義を示す。

クラス識別子型を例にした識別子に関する CONVERSION の定義

クラス識別子は、クラス識別子を構成するための型コンストラクタと文字列から構成される。2 つのクラス識別子があるときに、その比較の結果を真か偽で返す CONVERSION を定義する。

- 比較する要素が等しい場合

$$\text{CLASSID "class1"} = \text{CLASSID "class1"} \tag{5.1}$$

  HOL の組込みの以下の RULE である REFL を用いて、上の項の左辺を変形して以下の定理を導くことができる。これは、左辺と右辺が等しいことにより可能となっている。

REFL

$$\vdash t_1 \ = \ t_1$$

その結果、以下の定理を導くことができる。

$$\vdash \text{CLASSID "class1"} = \text{CLASSID "class1"}$$

さらに、EQT_INTRO を用いて、上の定理を変形する。

EQT_INTRO

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

その結果、

$$\vdash \text{CLASSID "class1"} = \text{CLASSID "class1"} = T$$

を導くことができる。これより、クラス識別子同士が等しいときの比較を導くことができた。

- クラス識別子同士が異なるとき、
以下の項に関する比較を行う。

$$\text{CLASSID "class1"} = \text{CLASSID "class2"} \tag{5.2}$$

クラス識別子の定義をした際に示した、以下のクラス識別子の一意性の定理を用いる。

$$\vdash \forall s \ s'. \ (\text{CLASSID s} = \text{CLASSID s}') = (\text{s} = \text{s}')$$

この定理を、比較するクラス識別子の値である"class1"と"class2"で具体化することで、以下の項を導くことができる。

$$\vdash (\text{CLASSID "class1"} = \text{CLASSID "class2"}) = (\text{"class1"} = \text{"class2"}) \tag{5.3}$$

ここで、仮定を持つ定理を導く RULE である ASSUME を利用する。

ASSUME

$$t \vdash t$$

ASSUME と、5.2 より、以下の定理を導く。

$$(\text{CLASSID "class1"} = \text{CLASSID "class2"}) \vdash (\text{CLASSID "class1"} = \text{CLASSID "class2"}) \tag{5.4}$$

更に 3 段論法を行う RULE である EQ_MP を利用する。

EQ_MP

$$\frac{\Gamma_1 \vdash t_1 = t_2 \quad \Gamma_2 \vdash t_2}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

EQ_MP を用いて 5.4と 5.2より、以下の定理を導くことができる。

$$(\text{CLASSID "class1"} = \text{CLASSID "class2"}) \vdash (\text{"class1"} = \text{"class2"}) \tag{5.5}$$

ここで、文字列の公理系より文字列の比較に関する CONVERSION を利用する。文字列の比較に関する CONVERSION は文字列の等式の項をとり、その比較を真か偽で返す CONVERSION として定義されている。

文字列に関する CONVERSION をクラス識別子を構成する文字列の等式に適用すると、以下の定理が得られる

$$\vdash (\text{"class1"} = \text{"class2"}) = F \tag{5.6}$$

もう一度 EQ_MP を用いて、5.5と 5.6から、

$$(\text{CLASSID "class1"} = \text{CLASSID "class2"}) \vdash F \tag{5.7}$$

が得られる。ここで、DISCH を用いる。

$$\frac{\Gamma, t_1 \vdash t_2}{\Gamma \vdash t1 \Rightarrow t_2}$$

これにより、5.7から以下の定理を導くことができる。

$$\vdash (\text{CLASSID "class1"} = \text{CLASSID "class2"}) \Rightarrow F$$

さらに、NOT_INTRO より、

NOT_INTRO

$$\frac{\Gamma \vdash t \Rightarrow F}{\Gamma \vdash \neg t}$$

以下の定理をえる。

$$\vdash \neg(\text{CLASSID "class1"} = \text{CLASSID "class2"})$$

さらに EQF_INTRO を用いることで、

EQF_INTRO

$$\frac{\Gamma \vdash \neg t}{\Gamma \vdash t = F}$$

$$\vdash (\text{CLASSID "class1"} = \text{CLASSID "class2"}) = F$$

となり、クラス識別子の比較に関して、偽を示す定理を導くことができた。

以上のことから、クラス識別子の比較に関して、真か偽かを導く CONVERSION を定義するこ とができた。

簡単にまとめると、識別子の比較に関しては、識別子の一意性に関する定理を利用して定義する。識別子型の要素は対応する文字列型の要素と 1 対 1 に対応することと、文字列型の値に関する CONVERSION を利用することで、識別子の値に関する CONVERSION を定義できる。つまり、識別子はそれ自身の構成要素の文字列に 1 対 1 に対応する。そして、HOL では文字列のレベルでの比較の仕組みが提供されているので、文字列での比較の結果を対応する識別子の結果として利用するわけである。

クラス識別子を例に識別子の CONVERSION の定義を示したが、他の識別子についても、同様の定義で CONVERSION を定義できる。

5.2.2 式/写像の実装

FO∀M における式は、対象システムの構造を表現するもので、識別子から構成される。また、式を構成している識別子に関して操作等を行うことが可能である必要があるので、そのような操作が可能な型として FO∀M の式を HOL 上に定義する必要がある。

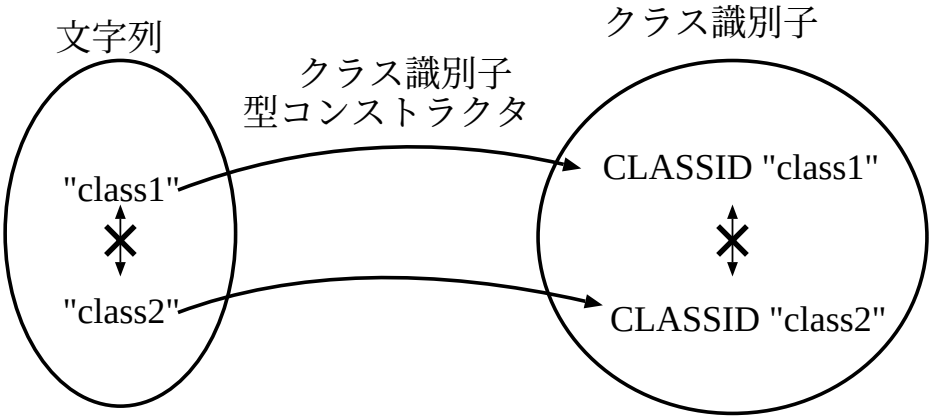

図 5.4: 識別子の CONVERSION

各基本モデルは、自身の構造を表現する複数の式を持っている。式は各基本モデルを構成する要素から構成されている。よって式は、その式を構成する識別子型と型コンストラクタを用いて実装する。そして、自身の型をもち、他の式とは明確に区別されるよう実装する。

継承式を例にした、式に関する型の定義

以下にオブジェクトモデルの継承式を例にして実際の式を表現する型に関する定義を示す。継承式は親クラスと子クラスを表すクラス識別子から構成されるので、クラス識別子から継承式型を構成するコンストラクタに関する定義をする必要がある。継承式を構成するための型コンストラクタは以下の一連の定義により定義できる。

まず、継承式という型を構成する要素に関する定義を行う。以下は継承式のを構成するための最下位レベルでのコンストラクタの役割を果たすラムダ関数の定義である。

```
val MK_INHER_DEF =
  new_definition("MK_INHER_DEF",
                 --`MK_INHER (x:ClassID) (y:(ClassID list))
                    = \a b.(a=x)/\(b=y)`--);
```

new_definition は引数として名前と HOL の項をとり、項で表現されている定義を HOL の

公理系の中に引数で取った名前で保存する。この定義により、以下の公理が定義される。

$$\vdash \forall \mathrm{x\ y.\ MK_INHER\ x\ y} = (\lambda \mathrm{a\ b.(a = x) \wedge (b = y)}) \tag{5.8}$$

この公理により、MK_INHER が定義される。このラムダ関数は後に定義する継承式のトップレベルのコンストラクタとなるラムダ式がとる引数を継承式の要素とすり合わせるために用いられる。

上で定義された最下位レベルで定義される型コンストラクタによって作られる集合に含まれる要素かどうかを調べるラムダ式を以下に定義する。これは、上の定義で定義した要素が成立するかを証明する際に用いられる定理である。

```
val IS_INHER_DEF =
  new_definition("IS_INHER_DEF",
                 --`IS_INHER p =
                    ?(x:ClassID) (y:(ClassID list)). p =
                    MK_INHER x y`--);
```

これにより、以下の定理がつくられ、生成される公理は ML 変数 IS_INHER_DEF に保存される。この結果定義されるラムダ関数 IS_INHER は、MK_INHER で定義される集合に対して bool 値を返す。

$$\vdash \forall \mathrm{p.\ IS_INHER\ p} = (\exists \mathrm{x\ y.\ p = MK_INHER\ x\ y}) \tag{5.9}$$

以下の証明で、上で定義した継承式を定義するため用いるラムダ関数から生成される集合が存在することを証明する。この定義は、継承式型を定義する際に用いられる。

```
val INHER_EXISTS =
  prove(
        --`?p.IS_INHER (p:ClassID->(ClassID list)->bool)`--,
           EXISTS_TAC (--`MK_INHER (x:ClassID)(y:(ClassID list))`--)
           THEN REWRITE_TAC[MK_INHER_DEF,IS_INHER_DEF]
           THEN EXISTS_TAC (--`x:ClassID`--)
           THEN EXISTS_TAC (--`y:(ClassID list)`--)
           THEN REWRITE_TAC[] );
```

この関数適用の結果、以下の定理を得る。

$$\vdash \exists p.\ \mathrm{IS_INHER}\ p \tag{5.10}$$

これは、以下の一連の証明と等価で、HOL の TACTIC と TACTICAL を組み合わせて証明を一度におこなっている。

これ以降の証明では、証明に用いられている要素に関して、以前の定義より型が明らかなものに関しては型の説明は省略する。

証明

証明は、まず以下の項を証明のゴールとして与える。

$$\exists\ p.\mathrm{IS_INHER}\ p \tag{5.11}$$

p を (MK_INHER x y) で具象化する。この項に関しては、5.8でその存在が既に定義されている。その結果、以下を得る。

$$\mathrm{IS_INHER}\ (\mathrm{MK_INHER}\ x\ y) \tag{5.12}$$

つぎに、5.8の定理を書き換えに用いて、項の書き換えを行い以下を得る。

$$\mathrm{IS_INHER}\ (\lambda a\ b.\ (a = x) \wedge (b = y)) \tag{5.13}$$

さらに、5.9の定理を書き換えに用いる。

$$\exists\ x'\ y'.(\lambda a\ b.(a = x) \wedge (b = y)) = \mathrm{MK_INHER}\ x'\ y' \tag{5.14}$$

ここで、x' と y' をそれぞれ ClassID 型の x と ClassID 型のリストである y で具象化する。

$$(\lambda a\ b.(a = x) \wedge (b = y)) = \mathrm{MK_INHER}\ x\ y \tag{5.15}$$

ここで 5.8を用いることで、証明できた。この定理は、IS_INHER で定義される集合が存在していることを示す。

以上の公理と定理を用いて、継承式型のための定理を定義する。

継承式型の定義には HOL の組込み ML 関数 new_type_definition を以下のように適用する。

```
val InherExp =
    new_type_definition{
              name = "InherExp",
              pred = --`IS_INHER:(ClassID->(ClassID list)->bool)->bool`--,
              inhab_thm = INHER_EXISTS};
```

この関数は、pred の引数を満たすような存在があることが inhab_thm の引数として与えられた定理で証明されているとき、その存在をを name の引数で与えた名前で HOL の型を定義する HOL 組込みの ML 関数である。この関数を実行した結果、以下の定理が自動的に証明され、継承式 (InherExp) 型が定義される。継承式型の定義には、上で示された 5.9、5.10の定理を用いている。

$$\vdash \exists \text{rep. TYPE_DEFINITION IS_INHER rep}$$

ここで現われている TYPE_DEFINITION は、新しい型が引数でとる述語で定義される集合の部分集合であることを保証する HOL の組込みの述語である。

この結果、継承式型 InherExp は今までに定義されている型から定義することができる型であることが保証され、HOL の型として以後利用可能になる。

次は、継承式型を表現する項の定義を行う。型の定義とその表現を別々に定義するのは、後で示す型を構成する要素へのアクセスに関する定義を簡単に行うためである。

以下の定義は、型の定義とその型を構成するためのラムダ関数を対応させるために必要な定義である。

```
val REP_InherExp =
  new_definition("REP_InherExp",
                 --`REP_InherExp  =
                    @rep:InherExp->(ClassID->(ClassID list)->bool).
                    (!p' p''. (rep p' = rep p'') ==> (p' = p'')) /\
                    (!p. IS_INHER (p:ClassID->(ClassID list) ->bool)
                     = (?p'. p = rep p'))`--);
```

この結果として、ML 変数 REP_InherExp に以下の定理が保存される。この定理は、次で定義する継承式型を構成するためのラムダ関数と、実際の継承式型を対応させるための補

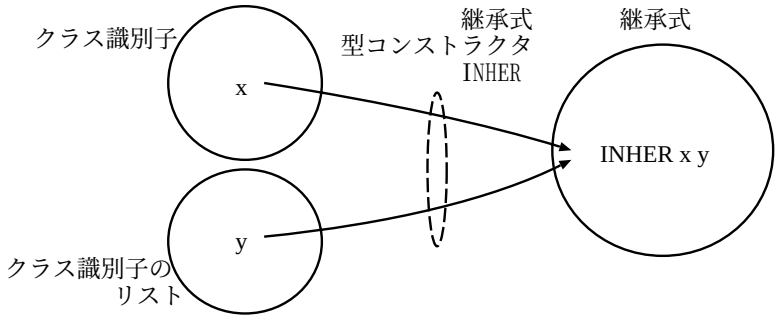

図 5.5: 継承式を構成する要素からの式への写像

助定理となっている。

$$\vdash \text{REP_InherExp} \;=\; (\epsilon\, rep.(\forall\, \mathrm{p}'\ \mathrm{p}''.\ (\mathrm{rep}\ \mathrm{p}' = \mathrm{rep}\ \mathrm{p}'') \Rightarrow (\mathrm{p}' = \mathrm{p}'')) \wedge (\forall\, \mathrm{p}.\mathrm{IS_INHER}\ \mathrm{p} = (\exists\, \mathrm{p}'.\ \mathrm{p} = \mathrm{rep}\ \mathrm{p}')))$$

以上のような定義をふまえて、継承式型を構成するラムダ関数 INHER の定義を行う。この関数は、継承式型を HOL の項で表現するために用いられる。

```
val INHER_DEF =
  new_definition("INHER_DEF",
                 --`INHER (x:ClassID) (y:(ClassID list))
                    = @p. REP_InherExp p
                    = MK_INHER x y`--);
```

この結果、以下の公理が作られる。これは、継承式を構成する要素から継承式型を構成するための型コンストラクタ INHER の定義といえる。

$$\vdash \forall\, \mathrm{x\ y}.\mathrm{INHER\ x\ y} = (\epsilon\, \mathrm{p.\ REP_InherExp\ p} = \mathrm{MK_INHER\ x\ y})$$

このように、実際の型の定義と型を表現するための定義を分けることで、型の表現を関数表現に写像して定義することができる。そして、このように構築された型は、型を構成する要素に関する定義を簡単に行える。

以上の定義より以下の継承式型の項を HOL で扱うことが可能になる。

```
(--` INHER (CLASSID "parent")
                  [(CLASSID "child1");(CLASSID "child2")] `--)
```

式型の要素に関する定義

式型を関数を利用して構成するよう定義したことで、式から式の構成要素を求めるための関数の定義を簡単に行うことが可能になる。

継承式型を例にした式の要素に関する定義

以下は継承式から導かれる親クラスに関する関数の定義である。これは、継承式型の表現を実際の継承式型の定義と分ける形で定義したことにより、簡単に定義できる。

```
val INHER_PARENT_DEF =
  new_definition("INHER_PARENT_DEF",
                --`INHER_PARENT (p:InherExp) =
                   @x. ?y. MK_INHER x y =
                   REP_InherExp p`--);
```

$$\vdash \forall\, \mathrm{p}\ \mathrm{INHER_PANRENT}\ \mathrm{p} = (\epsilon\, \mathrm{x}.\ \exists\, \mathrm{y}.\ \mathrm{MK_INHER}\ \mathrm{x}\ \mathrm{y} = \mathrm{REP_InherExp}\ \mathrm{p})$$

上の定義を用いて、定義した継承式型から親クラスを表すクラス識別子を求める関数を継承式に適用した結果が、継承式の親クラスと等しいことを証明できる。この証明はHOL の組込みの ML 関数で自動的に行うことができる。

```
val INHER_PARENT =
  save_thm ("INHER_PARENT",
             mk_thm([], --`!(x:ClassID)(y:ClassID list).
                        INHER_PARENT(INHER x y) = x`--));
```

$$\vdash \forall\, \mathrm{x}\ \mathrm{y}\ \mathrm{INHER_PARENT}\ (\mathrm{INHER}\ \mathrm{x}\ \mathrm{y}) = \mathrm{x} \tag{5.16}$$

子クラスを表す要素に関しても、同様の方針で定義ができる。

```
val INHER_CHILD_DEF =
  new_definition("INHER_CHILD_DEF",
                --`INHER_CHILD (p:InherExp) =
                   @y. ?x. MK_INHER x y =
                   REP_InherExp p`--);
```

$$\vdash \forall \text{ p INHER_CHILD p} = (\epsilon \text{ y. } \exists \text{ x. MK_INHER x y} = \text{REP_InherExp p})$$

```
val INHER_CHILD =
  save_thm ("INHER_CHILD",
            mk_thm([], --`!(x:ClassID)(y:ClassID list).
                          INHER_CHILD(INHER x y) = y`--));
```

$$\vdash \forall \text{ x y INHER_CHILD (INHER x y)} = \text{y} \tag{5.17}$$

これらの定理により、継承式型を構成する要素に対して、継承式型から関数適用の形でアクセスすることができる。

このように、継承式型の表現を関数で定義して、実際の型の定義から分離して定義したことによりその要素にアクセスする関数の定義を簡単に証明することができる。

そして、これらの定理を利用することで継承式型の要素を用いる証明において、証明を簡単に行うことができる。

上で示した定理を用いて、継承式の等価性に関する定理を定義する。継承式型の比較は、継承式型を構成する要素がそれぞれ等しいかどうかを調べることで行うことができる。上で証明した、親クラスや子クラスに関する定理を用いて書き換えを行いそれを証明する。

```
val INHER_EQ = store_thm("INHER_EQ",
   --`(INHER (x:ClassID) (y:ClassID list) = (INHER a b)) =
      ((x=a) /\ (y=b))`--,
      EQ_TAC THENL
      [DISCH_THEN (fn th =>
       REWRITE_TAC [REWRITE_RULE [INHER_PARENT]
                   (AP_TERM (--`INHER_PARENT:InherExp->ClassID`--) th),
                    REWRITE_RULE [INHER_CHILD]
                   (AP_TERM (--`INHER_CHILD:InherExp->(ClassID list)`--) th)]),
       STRIP_TAC THEN ASM_REWRITE_TAC[]]);
```

この定理を書き換え規則に利用することで、継承式同士の比較を簡単に証明できるようになる。式の比較を識別子の比較にすることができる。識別子の比較は前に定義した識別子の CONVERSION を利用して行うことができるので、結果として式の等価性に関する証明が可能となる。

$$\vdash (\mathrm{INHER\ x\ y} = \mathrm{INHER\ a\ b}) = ((\mathrm{x} = \mathrm{a}) \wedge (\mathrm{y} = \mathrm{b})) \tag{5.18}$$

この定理を用いることにより、式が等しいということは、式の構造が等しく、式を構成している各識別子のそれぞれが等しいことを示すことで求めることが可能になる。識別子の比較に関しては前に定義した識別子の CONVERSION を用いることで可能である。以上より、継承式型に関する等価性を示すことができた。

他の式型に関しても同様の方針で、型を構成する要素に関する定義や等価性に関する定義を行うことができる。

写像の定義

FO∀M では、式と式に対応する識別子との写像により、対象システムの構造を表現する要素を定義する。例えば、継承識別子と継承式の写像により、1 つの継承関係を定義する。

FO∀M では写像は識別子と式とのマッピングとして定義されているが、検証フレームワーク上の表現としては、主に実装の都合から写像も型の一つとして定義する。そして、写像を構成する要素の間の関係は写像型の上に定義される性質として、論理式で定義することする。写像型の定義に関しては、構成要素が識別子と式となるだけで、式型と同様の実装方針で定義できる。

継承識別子と継承式を例にした、継承式の写像に関する定義

以下に継承識別子と継承式を例として、写像を表現する型に関する定義を示す。継承式には、それに対応づけられる継承識別子が写像で定義されている。対象システムにおける 1 つの継承関係を表現する。

HOL 上では以下の項が継承式の写像型として用いられる。

```
(--` INHER_MAP (INHERID "inher1")
               (INHER (CLASSID "parent")
                      [(CLASSID "child1");(CLASSID "child2")]) `--)
```

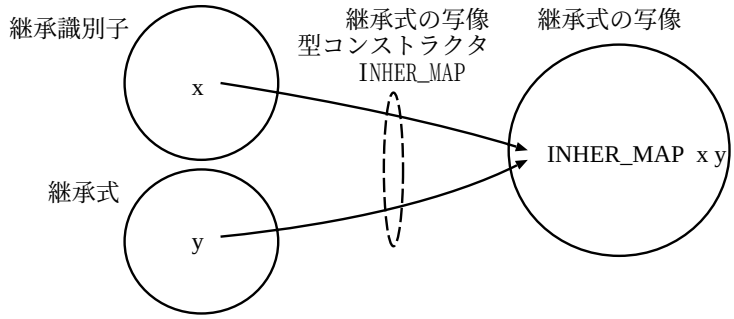

図 5.6: 継承式をの写像を構成する要素からの継承式の写像型への写像

以上のように、FO∀M の理論で扱っているデータ構造のそれぞれに対して、その構築方法を示した。また、それぞれのデータ型における等価性の定義や、補助定理を示した。解説では各データ型に関して 1 つ例をあげてその構築方法を示したが、例で示さなかった識別子や式などに関しても、同様の方法で構築することが可能である。よって、FO∀M の理論で扱っているデータ構造に関する定義をすべて行うことが出来た。

5.2.3 規則/制約等の実装と検証の補助のための定義

FO∀M では識別子や式、写像などの対象システムの構造に関するデータ構造を表す要素に対して、その上に成立する性質として規則や制約を定義している。よって、それらの定義に関しては、これまでで定義した FO∀M のデータ構造をもとに定義される公理や定理として実装する。

また、性質に関して明確に定義されていない場合でも、暗黙の了解として成立している性質に関しては、その性質を明らかにして定義する必要がある。

継承式を例にした継承式上に定義される性質の定義　継承式によってクラス識別子間の継承関係が定義されると、そこにクラス識別子間の継承関係における順序関係が定義される。以下の定義で、その順序関係に関する定義を行っている。

DEFINE_INHER は継承式型をとる述語で、ここでは継承式として有効であるのは、定義されているものだけであることを示すために使っている。式が有効であるという議論は、次節のモデル情報において詳しく説明を行う。

```
val INHER_ORDER_DEF =
```

```
new_definition("INHER_ORDER_DEF",
               --`INHER_ORDER (c:ClassID) (p:ClassID) =
                  ?(iexp:InherExp). DEFINE_INHER iexp /\
                  (MEMBER c (INHER_CHILD iexp)) `--);
```

$$\vdash \forall \mathrm{c\ p.INHER_ORDER\ c\ p} \;=\; (\exists \mathrm{iexp.DEFINE_INHER\ iexp} \wedge (\mathrm{MEMBER\ c\ (INHER_CHILD\ iexp)})$$

この定理は、あるクラス識別子間において継承の順序関係が成立するのは、順序関係求める 2 つのクラス識別子に関して、そのクラス識別子を含む継承式が成立していて、かつその 2 つのクラス識別子が継承式のそれぞれ親クラスと子クラスを示す識別子であるときであることを示している。

上の定理では、継承式から直接順序関係が導かれるクラス間のみ順序関係を示すことができる。実際の継承関係では継承か複数回行われる場合もあり、その場合でも継承関係の系列に含まれるクラスの間で順序関係が成立しなければならない。これは、継承関係の順序関係において推移律が成立すると考えられるので、それを満たすための定理を作る必要がある。

上で定義した継承関係における順序関係に関する定理を利用して、以下に継承関係の順序関係における推移律を定義する。

```
val INHER_TRANS =
  new_definition("INHER_TRANS",
                 --`!INHER_ORDER:(ClassID->ClassID->bool).
                    INHER_TRANS INHER_ORDER =
                    !a b c:ClassID.
                       (((INHER_ORDER a b) /\ (INHER_ORDER b c)) =
                    INHER_ORDER a c)`--);
```

$$\vdash \forall \mathrm{INHER_ORDER.\ INHER_TRANS\ INHER_ORDER} \;=\; (\forall\ \mathrm{a\ b\ c.\ INHER_ORDER\ a\ b} \wedge \mathrm{INHER_ORDER\ b\ c} = \mathrm{INHER_ORDER\ a\ c})$$

このように、FO∀M を構成するデータに関する性質を 1 つ 1 つ定理として定義していく。他の制約や規則に関しても基本的に同様の方針で定義していく。まず、データ構造をとり、その制約条件を定義し、必要なら更にその定義を用いて必要な制約や規則を満たす定理を定義していく。そのよう定義を繰り返すことで、最終的に FO∀M で定義されているデータ構造上に定義される規則や制約を HOL 上に定義することができる。

以下では、更にこの定義を実際の検証に用いる際に、簡単に利用するための定義を行う。

上の定義では HOL のシステム上の都合から直接項の書き換えに用いることができない。よって、証明の書き換え規則で用いることが可能になるよう上の定義を具体化して、変形しておく。

```
val INHER_TRANS_RULE_DEF =
  SPECL [(--`a:ClassID`--),(--`b:ClassID`--),(--`c:ClassID`--)]
  (UNDISCH
   (fst (EQ_IMP_RULE
         (SPEC (--`INHER_ORDER:(ClassID->ClassID->bool)`--) INHER_TRANS))));
```

$$\text{INHER_TRANS INHER_ORDER} \vdash \quad (\text{INHER_ORDER a b} \wedge \text{INHER_ORDER b c} = \text{INHER_ORDER a c})$$

これにより、継承関係の順序関係における推移律を示す定理を証明の書き換えに用いることができる形にすることができた。

これで、継承関係の順序関係に関する定義ができたが、さらに実際の証明に用いる際に有効な TACTIC を作る。

```
val INHER_TRANS_TAC =
  IMP_RES_TAC INHER_TRANS_RULE_DEF;
```

IMP_RES_TAC は、この TACTIC で証明された結果を証明の仮定に追加して更に証明をすすめる TACTIC である。ゴールの書き換えは、この TACTIC を用いておこなう書き換えが行えない状態になるまで続く。

この定義により、継承関係の順序関係に対して推移律を用いて証明された順序関係を次々に仮定に追加することで継承の順序関係に関する変換を行う TACTIC を作ることができる。

これまでで定義したように、FO∀M の理論を構成するデータ構造に関する性質を、そのデータ型をもとにして定義される述語で定義する。。他の性質に関しても、定義の方法は異なるが同様の方針で定義していく。

5.3 モデル情報

対象システムのモデルの検証をおこなうためには、検証フレームワークにそのモデルの情報を与える必要がある。

この、検証フレームワークに与えられる、検証するシステムに依存したモデルの情報をモデル情報とする。モデル情報は、各識別子、写像、式の型をとる述語を定義して、論理式の形で検証フレームワークに与える。

証明の際には、モデル情報として与えられたもののみが成立すると定義する。よって、対象システムの性質に関しては、全てモデル情報と FO∀M の理論の公理系から導くことになる。そのことから、モデル情報から導くことが出来ない性質は、もとのモデルで成立しない性質であるといえる。

5.4 検証の方針

構築した検証フレームワークを用いて対象システムのモデルに関する検証を行う場合は、モデル情報を入力として、検証フレームワークで定義されている公理を利用して検証を行う。

検証は、ますモデル情報を前提条件として与える。そして、対象システムにおいて検証したい性質を証明のゴールとして与える。前提条件から、検証フレームワークで定義されている公理群を用いてゴールとして与えられた性質が証明できれば、その性質が対象システムのモデルにおいて成立していることを保証できる。

つまり、与えられたモデル情報から目的とする性質が検証できれば、その性質はモデル上で成立しているといえる。

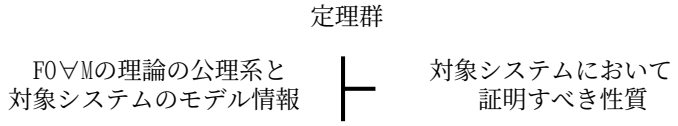

図 5.7: FO∀M で記述されたモデルの検証方針

5.5 実装のまとめ

以上のように FO∀M の構造の分析に従い、FO∀M の理論と FO∀M で用いられるデータ構造の定義を行った。また、それぞれのデータが持つ性質をデータ構造上に成立する定理として定義した。実装の例では、オブジェクトモデルの構成要素を例にとり解説したが、基本的に他のモデルでも同様の構造を持つため、同様の方針で構築できる。最終的に、FO∀M の理論を HOL 上に構築でき、FO∀M のモデルに関する検証フレームワークが実装できた。

第 6 章

検証フレームワークを用いた検証例

この章では、簡単な例題を用いて、構築した検証フレームワークにおける、モデルの検証の例を示す。モデルの表現に関しては、FO∀M の定義は青木の FO∀M の仕様書に従う。

6.1 例題

例題として、簡単な電気スタンドを考える。その仕様は以下に従う。

- スイッチを 1 つとライトを 1 つ持つ。
- スイッチはトグル式で押すごとにオン状態とオフ状態が切り替わる。
- スイッチがオン状態になるとライトが付き、オフ状態になるとライトが消える。

この電気スタンドの分析モデルを次節以降で示す。各分析モデルには OMT の表記と、FO∀M の表記、そして検証フレームワークでの表記で示す。

6.1.1 基本オブジェクトモデル

FO∀M の表記による基本オブジェクトモデル

$$light = (Stand, (Blub, Switch))$$

$$Stand = ()$$

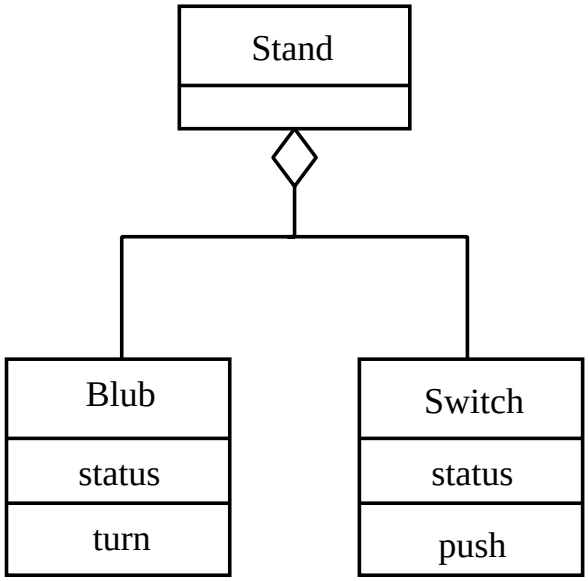

図 6.1: OMT の記法で記述したオブジェクトモデル

$$Bulb = ((status), (turn))$$

$$Switch = ((status), (push))$$

検証フレームワーク上の表記による基本オブジェクト モデル

```
(--`AGGR_MAP  (AGGRID "light")
              (AGGR (CLASSID "Stand")
                   [ (CLASSID "Bulb");
                     (CLASSID "Switch")]) `--)
(--`CLASS_MAP (CLASSID "Stand")
              (CLASS [ ]
                     [ ]) `--)
(--`CLASS_MAP (CLASSID "Bulb")
              (CLASS [ (ATTRID "status") ]
                     [ (FUNCID "turn") ]) `--)
(--`CLASS_MAP (CLASSID "Switch")
              (CLASS [ (ATTRID "status") ]
                     [ (FUNCID "push") ]) `--)
```

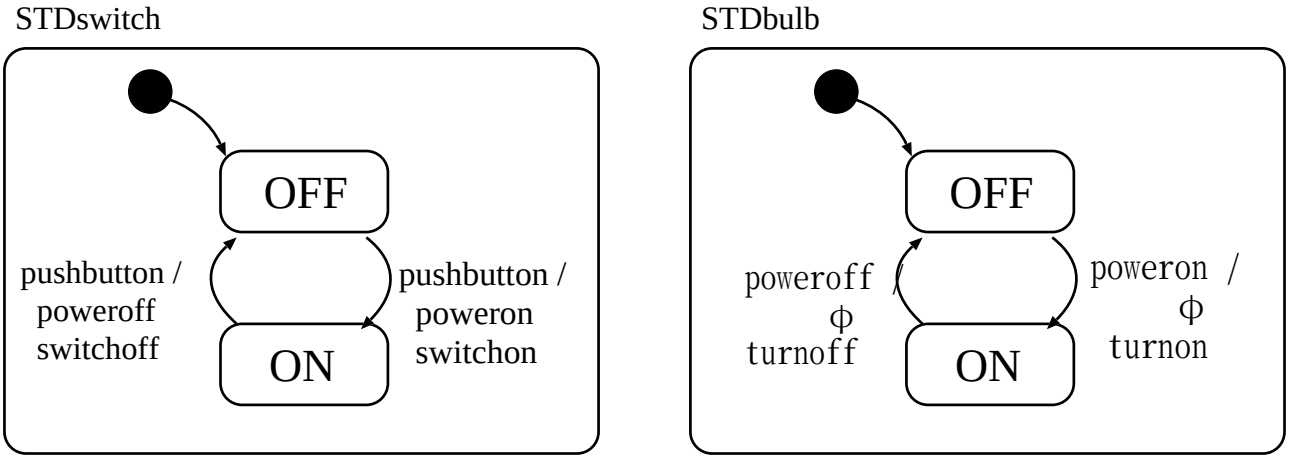

図 6.2: OMT の記法で記述した動的モデル

6.1.2 基本動的モデル

FO∀M の表記による基本動的モデル

$$
\begin{aligned}
STDswitch &= (State, Evt_{in}, Evt_{out}, Act, Cond, Trans, OFF) \\
where \quad & State = \{ON, OFF\} \\
& Evt_{in} = \{pushbutton\} \\
& Evt_{out} = \{poweron, poweroff\} \\
& Act = \{switchon, switchoff\} \\
& Cond = \{\} \\
& Trans = \\
& \{(OFF \rightarrow ON|pushbutton[]/poweron\ switchon]) \\
& (ON \rightarrow OFF|pushbuttoff[]/poweroff\ switchoff])\}
\end{aligned}
$$

$$
\begin{aligned}
STDbulb &= (State, Evt_{in}, Evt_{out}, Act, Cond, Trans, OFF) \\
where \quad & State = \{ON, OFF\} \\
& Evt_{in} = \{poweron, poweroff\} \\
& Evt_{out} = \{\}
\end{aligned}
$$

$$Act = \{turnon, turnoff\}$$

$$Cond = \{\}$$

$$Trans =$$

$$\{(OFF \rightarrow ON|poweron[]/\phi\ turnon])$$

$$(ON \rightarrow OFF|poweroff[]/\phi\ turnoff])\}$$

検証フレームワークの表記による基本動的モデル

```
(--‘ STD_MAP (STDID "STDswitch")
             STD [(STATEID "ON");(STATEID "OFF")]
                 [(EVENTID "bushbutton")]
                 [(EVENTID "poweron");(EVENTID "poweroff")]
                 [(ACTID "switchon");(ACTID "switchoff")]
                 []
                 [(ST (STATEID "OFF") (STATEID "ON")
                      (DEF_EVT (EVENTID "pushbutton"))
                      (CONDID "") (DEF_EVT (EVENTID "poweron"))
                      (ACTID "switchon"));
                  (ST (STATEID "ON") (STATEID "OFF")
                      (DEF_EVT (EVENTID "pushbutton"))
                      (CONDID "") (DEF_EVT (EVENTID "poweroff"))
                      (ACTID "switchoff"))]
                  (STATEID "OFF") ‘--)

(--‘ STD_MAP (STDID "STDbulb")
             STD [(STATEID "ON");(STATEID "OFF")]
                 [(EVENTID "poweron");(EVENTID "poweroff")]
                 []
                 [(ACTID "turnon");(ACTID "turnoff")]
                 []
                 [(ST (STATEID "OFF") (STATEID "ON")
```

```
    (DEF_EVT (EVENTID "poweron"))
    (CONDID "") (EMPTYEVENT)
    (ACTID "turnon"));
(ST (STATEID "ON") (STATEID "OFF")
    (DEF_EVT (EVENTID "poweroff"))
    (CONDID "") (EMPTYEVENT)
    (ACTID "turnoff"))]
(STATEID "OFF") `--)
```

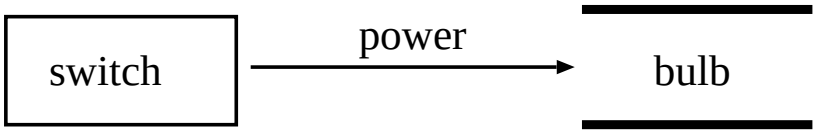

図 6.3: OMT の記法で記述した機能モデル

6.1.3 基本機能モデル

FO∀M の表記による基本機能モデル

$$((switch, bulb), power)$$

検証フレームワーク上の表記による基本機能モデル

```
(--‘ DATAFLOW (PROCESS_NODE (PROCESSID "switch"))
              (STORE_NODE   (STOREID "bulb"))
              (DATAID "power") ‘--)
```

6.1.4 統合モデル

FO∀M の表記による統合モデル。

状態遷移図の統合写像

$$Switch \mapsto STDswitch$$

$$Bulb \mapsto STDbulb$$

アクション式

$$Switch.status := Switch.push(Switch.status)$$

$$Bulb.status := Bulb.turn(Bulb.status)$$

アクション式の統合写像

$$switchoff \mapsto Switch.status := Switch.push(Switch.status)$$

$$switchon \mapsto Switch.status := Switch.push(Switch.status)$$

$$turnon \mapsto Bulb.status := Bulb.turn(Bulb.status)$$

$$turnoff \mapsto Bulb.status := Bulb.turn(Bulb.status)$$

アクターの統合写像

$$switch \mapsto ((Switch.status), (Switch.push))$$

データストアの統合写像

$$bulb \mapsto ((Bulb.status), (Bulb.turn))$$

検証フレームワークの表記による統合モデル

状態遷移図の統合写像

```
(--‘ ISTD (CLASSID "Switch") (STDID "STDswitch") ‘---)
(--‘ ISTD (CLASSID "Bulb") (STDID "STDbulb") ‘---)
```

アクション式

```
(--‘ ASSIGN_AO (CA (CLASSID "Switch") (ATTRID "status"))
               (REF_FO (CF (CLASSID "Switch") (FUNCID "push"))
                       (REF_AO (CA (CLASSID "Switch") (ATTRID "status"))))
                  ‘--)
(--‘ ASSIGN_AO (CA (CLASSID "Bulb") (ATTRID "status"))
               (REF_FO (CF (CLASSID "Bulb") (FUNCID "turn"))
                       (REF_AO (CA (CLASSID "Bulb") (ATTRID "status"))))
                  ‘--)
```

アクション式の統合写像

```
(--‘ IACTION (ACTID "switchon")
             (ASSIGN_AO (CA (CLASSID "Switch") (ATTRID "status"))
               (REF_FO (CF (CLASSID "Switch") (FUNCID "push"))
                       (REF_AO (CA (CLASSID "Switch") (ATTRID "status")))))
                  ‘--)
(--‘ IACTION (ACTID "switchoff")
             (ASSIGN_AO (CA (CLASSID "Switch") (ATTRID "status"))
```

```
                (REF_FO (CF (CLASSID "Switch") (FUNCID "push"))))
                        (REF_AO (CA (CLASSID "Switch") (ATTRID "status")))))
                   `--)
(--` IACTION (ACTID "turnon")
     (ASSIGN_AO (CA (CLASSID "Bulb") (ATTRID "status"))
                (REF_FO (CF (CLASSID "Bulb") (FUNCID "turn"))
                        (REF_AO (CA (CLASSID "Bulb") (ATTRID "status")))))
                   `--)
(--` IACTION (ACTID "turnoff")
     (ASSIGN_AO (CA (CLASSID "Bulb") (ATTRID "status"))
                (REF_FO (CF (CLASSID "Bulb") (FUNCID "turn"))
                        (REF_AO (CA (CLASSID "Bulb") (ATTRID "status")))))
                   `--)
```

アクターの統合写像

```
(--` IPROCESS (PROCESSID "switch")
              ([(CAE  (CLASSID "Switch") (ATTRID "status"))],
               [(CAE  (CLASSID "Switch") (ATTRID "push"))]) `--)
```

データストアの統合写像

```
(--` ISTORE   (STOREID "bulb")
              ([(CAE  (CLASSID "Bulb") (ATTRID "status"))],
               [(CAE  (CLASSID "Bulb") (ATTRID "turn"))]) `--)
```

6.1.5 検証例

この節では、前節までで示されている分析モデルを例にして、検証フレームワークによる実際の検証手順を示す。-は検証フレームワークでのプロンプトを示す。

基本オブジェクトモデルにおける Bulb クラスが Stand クラスの部品クラスかどうかを検証する。

まず、集約関係式をモデル情報として証明の前提とし、証明したい事項を結論として証明のゴールを設定する。ここで、g は HOL の証明のゴールを設定する関数である。

```
- g `(DEFINE_AGGR (AGGR (CLASSID "Stand")
                   [ (CLASSID "Bulb");
                     (CLASSID "Switch")]) ==>
                   (IS_COMPONENT (CLASSID "Bulb") (CLASSID "Stand"))`;
val it =
  Status: 1 proof.
  1. Incomplete:
       Initial goal:
          (--` (DEFINE_AGGR (AGGR (CLASSID "Stand")
                   [ (CLASSID "Bulb");
                     (CLASSID "Switch")])) ==>
                   (IS_COMPONENT (CLASSID "Bulb") (CLASSID "Stand"))`--);
   : proofs
```

まず、証明のゴールの形を変形する。

```
- e STRIP_TAC;
OK..
1 subgoal:
val it =
  (--`IS_COMPONENT (CLASSID "Bulb") (CLASSID "Stand")`--)
  ----------------------------
      (--`DEFINE_AGGR
            (AGGR (CLASSID "Stand") [CLASSID "Bulb"; CLASSID "Switch"])`--)
   : goalstack
```

つぎに、集約関係の部品関係IS_COMPONENT の定義で書き換えを行う。

```
- e (REWRITE_TAC [IS_COMPONENT_DEF]);
OK..
1 subgoal:
val it =
  (--`?iexp.
```

```
      DEFINE_AGGR iexp /\ MEMBER (CLASSID "Bulb") (AGGR_COMP iexp)`--)
  ----------------------------
      (--`DEFINE_AGGR
            (AGGR (CLASSID "Stand") [CLASSID "Bulb"; CLASSID "Switch"])`--)
   : goalstack
```

モデル情報で示される情報より、項の具象化をおこなう。

```
- e (EXISTS_TAC
     (--`AGGR (CLASSID "Stand") [CLASSID "Bulb"; CLASSID "Switch"]`--));
OK..
1 subgoal:
val it =
  (--`DEFINE_AGGR
        (AGGR (CLASSID "Stand") [CLASSID "Bulb"; CLASSID "Switch"]) /\
      MEMBER (CLASSID "Bulb")
        (AGGR_COMP
          (AGGR (CLASSID "Stand") [CLASSID "Bulb"; CLASSID "Switch"]))`--)
  ----------------------------
      (--`DEFINE_AGGR
            (AGGR (CLASSID "Stand") [CLASSID "Bulb"; CLASSID "Switch"])`--)
   : goalstack
```

集約関係の部品関係の定義 AGGR_COMP とリストに対して、その要素かを調べる MEMBER の定義を用いて書き換えをおこなう。

```
- e (REWRITE_TAC [AGGR_COMP,MEMBER_DEF]);
OK..
1 subgoal:
val it =
  (--`DEFINE_AGGR
        (AGGR (CLASSID "Stand") [CLASSID "Bulb"; CLASSID "Switch"])`--)
  ----------------------------
```

```
      (--`DEFINE_AGGR
            (AGGR (CLASSID "Stand") [CLASSID "Bulb"; CLASSID "Switch"])`--)
   : goalstack
```

これで、仮定より証明できた。

```
- e (ASM_REWRITE_TAC []);
OK..
Goal proved.
|- (DEFINE_AGGR (AGGR (CLASSID "Stand")
                  [ (CLASSID "Bulb");
                    (CLASSID "Switch")])) ==>
                  (IS_COMPONENT (CLASSID "Bulb") (CLASSID "Stand"))
  : goalstack
```

よって、Bulb クラスが Stand クラスの部品クラスであるということは、モデル情報と、FO∀M 理論の定義より導くことができることを明らかにできた。

検証は、基本的のこの手順を繰り返すことで目的の定理を得る。

証明の書き換えに用いた定義

```
val IS_COMPONENT_DEF =
  new_definition("IS_COMPONENT_DEF",
                --`IS_COMPONENT (c:ClassID) (p:ClassID) =
                ?(iexp:AggrExp). DEFINE_AGGR iexp /\
                (MEMBER c (AGGR_COMP iexp)) `--);

val AGGR_COMP =
  save_thm ("AGGR_COMP", mk_thm([], --`!(x:ClassID)(y:^AGGR_CLASS list).
                              AGGR_COMP (AGGR x y) = y`--));

val MEMBER_DEF = new_recursive_definition
      {name = "MEMBER_DEF",
```

```
fixity = Prefix,
rec_axiom = list_Axiom,
def = --`(MEMBER (x:'a) [] = F) /\
(MEMBER (x:'a) (CONS (h:'a) t) =
 ((x = h) \/  MEMBER x t))`--};
```

第 7 章

考察

この章では構築した検証フレームワークに関する考察を行う。

7.1 検証フレームワークとして要求された機能に関する考察

- 構文チェック

  FO∀M で定義されているデータ構造は検証フレームワーク上ではそれぞれを型として実装している。そのために対象システムのモデルをモデル情報として検証フレームワークに与えることで静的な型チェックができる。よって、モデルに構文的な間違いがあれば、型に関して不正な部分があるはずなので、型チェックによってその誤りが発見できる。型チェックの処理は自動的に行われるので、ユーザによる特別な操作は必要としない。このことから、構文チェックに関して検証フレームワークは十分な機能を提供していると考えることができる。

- 一貫性チェック

  モデルが正しく構築されているかを検証するには、構文的な検証だけでは不十分である。そこで、モデルの論理的性質が FO∀M の理論との一貫性を満たすように構築されているかを検証する必要がある。

  モデルの一貫性チェックに関しては、与えられたモデル情報から、モデルの各構成要素に関して、その構成要素上に定義される規則や制約が満たされているかを、モデルの性質を表す定理を用いて検証することで確認できる。

検証フレームワークを用いることで、FO∀M の理論の公理系の利用は容易になる。よって、FO∀M の理論で定義されている定義を利用したモデルの一貫性に関する検証が容易になる。また、必要に応じて ML 関数を定義することで定型の検証に対しては簡単にその結果を得ることができる。

このように、検証フレームワークはモデルの一貫性の検証に対しても、十分な機能を提供することが確認できた。

- モデルの性質に関する理解とその検証

  構築した分析モデルに関して、そのモデルで成立している性質に関する検証をしたいという要求がある。

  検証フレームワークを用いることで、モデルの性質の検証をモデル情報を利用した証明によって行うことができる。その証明は FO∀M の公理系を用いた証明となる。証明の過程は 1 つ１つ公理や定理を用いながら進めることになるので、検証する性質に関して、それがどの規則から、どのように導かれるかを検証者が確認し、理解することが非常に容易になる。また、証明された性質は対象システムが満たしている性質であることを保証できる。

  このことから、検証フレームワークはモデルの性質に関する理解を深めるための機能を十分に提供するといえる。

FO∀M の検証フレームワークを用いることで、モデルの構文と一貫性の検証などによるモデルが正しく構築されているかの検証を計算機を用いて容易におこなうことが可能になった。

また、性質の検証をおこなうことで、モデルからどのような性質が導くことができるかと、その性質がどのように導かれるかを理解することが可能になる、検証フレームワークは、そのようなモデルの性質の検証のための機能を十分に提供することが確認できた。

以上のことから、実装した検証フレームワークは FO∀M を用いて分析されたモデルの検証支援環境として要求された項目に対して十分な機能を提供していること示すことができた。

7.2 検証フレームワークの利用に関する考察

検証フレームワークの利用に関する問題点も明らかになった。

- 中間定理の処理
  大きなシステムのモデルに関する検証を行う場合、検証における証明の過程で生成される中間定理が、膨大な数になることが確認できた。人間が検証を行う際には暗黙のうちに理解し、成立しているものと仮定してしまっている条件や性質にがある。しかし、検証フレームワークにおいては、公理系にもとづき、その定義にのっとり証明を進める。そのため、目的の検証のために人間が見れば一見当り前のような性質についても証明が必要な場合がある。

  このような中間定理は、目的とする証明からは無駄となる場合が多い。しかし、あらゆる状況でで無駄でとうわけではない。性質の検証の際の付加情報としての価値があり、それらの定理がモデルの性質の理解に関して重要な役割を果たすこともあるからである。また、モデルの誤りを発見するための手助けにもなる。

  ただ、中間の情報が重要ではなく、結果のみを知りたい場合には、目的の証明の妨げになることも確かである。

  この証明の過程における中間定理の制御が今後の課題としてあげられる。対処法としては、TACTIC や TACTICAL を用いて、定型の証明に関する自動証明のための補助定義をさらに充実させることが考えられる。

- 検証フレームワークの表記に関する問題
  FO∀M で扱っている複雑なデータ型を HOL 上に型として実装しているが、その表記に関してかなりデータに関する視認性が悪くなっている。特に、統合写像を表現するための型の表記に関してその傾向が強い。これは、構造を持つ型に対して内部で用いられている型コンストラクタの数が増えることに原因がある。

第 8 章

まとめ

8.1 まとめ

本研究では、オブジェクト指向方法論のための形式的モデルである FO∀M のための検証フレームワークを構築した。

始めに検証フレームワークとして必要な機能を分析し、実装の視点から FO∀M を再分析して、その理論の実装方針を決定した。次に、その実装方針にしたがい実装環境を決定し、FO∀M の理論を扱うことができる検証フレームワークを実装した。

また、実装した検証フレームワークの評価を行い、その有効性を確認した。

8.2 今後の課題

- 大きなシステムの検証では扱う定理が増えるので、証明の過程で現われる中間定理の数もそれにつれて増加する。その結果、 検証の際に目的の証明を行 うことが困難になる。よって、特定の証明を行うために有効な TACTIC を実装など、中間定理の制御を行う必要がある。

- FO∀M には、動作に関する意味についても定義されている。それらは、現在は検証フレームワーク上には未実装であるので、意味にに関する理論も実装する。

- FO∀M のモデル構築支援環境 [3] とのインターフェースを考慮して、修正を行う。

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鈴木正人助手、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] 古川順一: 形式的オブジェクト指向分析モデル FO∀M の構築法とその支援環境, Master's thesis, JAIST, 1996.

[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

[7] Winskel, G: The Formal Semantics of Programming Languages, The MIT Press, 1993.