2 階の論理によるプログラミング手法

河野真治

e-mail:kono@ie.u-ryukyu.ac.jp
Information Engineering, University of the Ryukyus
Nishihara-cyo 1, Okinawa, 903-01, Japan

October 20, 1996

1 高階論理、様相論理、メタ論理

まず最初にさまざまな論理について簡単に紹介しよう。

対象とその関連を真理値 (真か偽か) を使って表したのが論理である。論理は対象を指し示す変数を持つが、2 階の変数とは、対象を指し示すだけでなく、その対象の関連を表す式そのものを指し示す変数である。対象を表す変数は 1 階の変数と呼ぶ。さらに、2 階の変数を持つ論理式を指し示す変数は 3 階というように続いていく。n 階の変数を含む論理を n 階論理と呼ぶ。

真理値 (T と F) だけが対象の場合を命題論理と呼ぶ。命題論理の変数を簡単に命題変数と呼ぶ。対象にさまざまなものがあり、対象間の関係を表す述語の真理値を問題にする論理を述語論理という。この区別は n 階論理と独立な区別である。

真理値の基本的な計算は、$\wedge$ (かつ), $\vee$ (または), $\neg$ (でない) がある。これらの計算は真理値表によってきまるが、さらに、論理には、もともと真や偽と決まっている式 (公理) と、真偽の分かっている式から他の式の真偽を決める規則 (推論規則) がある。これらを直接扱ったり、変数と対象の結びつきそのものを扱うような論理はメタ論理と呼ばれる。

通常の論理では対象と真理値の結びつきを 1 対 1 とする。例えば時間によって真理値が変わると考えることもできる。この場合は真理値は時間と対象の二つから決まる。この対応を抽象化するための演算子として、$\diamond$ (いつか)、$\Box$ (いつも) が使われる。このような論理を様相論理という。特に、時間によって変化する状態に着目した論理を時相論理 (Temporal Logic) という。時相論理の命題変数は、その時刻で何が起こったか、起こらなかったかを表すイベントを表している。(他には、事象の時間的関係に主に着目する時制論理 (Tenth Logic) などがある) これらを含まない論理を特に区別する時は、古典論理と呼ぶことが多い。

1.1 オブジェクト指向言語の理論

オブジェクト指向言語には理論はないといわれるが、実際にはさまざまな方法により理論解析が行われている。関数型プログラミングの視点から作られた OBJ や、レコード型を中心とした解析、継承関係に注目した論理型プログラミングの理論などが知られている。しかし、開放型システムに対する理論研究は難しい。実際、不可能だという研究者もいる。

ここでは、メッセージ送信を 2 階の述語だと考えたオブジェクト指向プログラミング考察する。

オブジェクトは、インスタンス変数と呼ばれる状態変数と、メソッドと呼ばれる状態変数を変更する手続きの集合からなる。従って、オブジェクトの仕様は本質的に時相論理によって記述されるべきものである。

オブジェクト指向分析やオブジェクト指向設計は、データベース理論と、構造化設計の混合に近く、インスタンス変数の部分はデータベース理論にそって構築し、メソッドの部分は構造化設計の手法に従って設計されることが多い。しかし、本来、両方を同時に構築していくのがオブジェクト指向プログラミングである。

オブジェクト指向プログラミングの中心であるメッセージ送信は、オブジェクトを格納したインスタンス変数のオブジェクトにあるメッソドを呼ぶことで実現される。これを従来は、多相型関数と解釈することがおこなわれてきた。しかし、もしそうならオブジェクト指向プログラミングは、モジュール分割によるプログラミングと同等になってしまう。また、オブジェクトは状態を持っているので、厳密な意味での関数でもない。

オブジェクトは情報隠蔽の単位でもあるが、メッセージのやり取りは特に隠されてはいない。ここでメッセージのやり取りを時相論理のイベントと考える。オブジェクトのメソッドの仕様は 1 階の述語時相論理で記述される。メッセージのやり取りで、その仕様が実現されるわけである。このように考えると、オブジェクトは仕様そのものを指し示している 2 階の変数であることがわかる。

本来、オブジェクト指向設計は、仕様そのものや実装の変更に対処するための設計であり、オブジェクトの仕様がすべてきちんと決まっているならば、関数型言語あるいはモジュール型のプログラミング言語で記述すれば良い。オブジェクトには不定あるいは可変の部分が想定されている。それはパラメータによる変更などの 1 階の変数で表されるべきものではなく、2 階の変数で表現されるべきものである。

しかし、2 階で足りるのだろうか? メソッドの引き数として渡されるのは、数値や文字などの 1 階の対象物ではなく、オブジェクトそのものも渡される。また、オブジェクトはさまざまな方法で生成される。これらの操作は、オブジェクトそのものを対象にしているので、3 階以上の仕様だと考えられる。一つの方法は、高階論理をそのまま採用することだが、ここではもっと別な方法を考えてみたい。

本来、オブジェクトはライフサイクルを持っているのであり、短期間に多量に生成消滅するものではない。そのよう

な生成消滅の激しいものは、オブジェクトとしてではなく、数値、文字、文字列、リスト、配列などのように基本的な1階の対象物であるとする。あるオブジェクトが生成されてから消滅するまでは、それらが関連するオブジェクトは一種の定常状態にある。実際に検証やプログラミングの対象になるのは、この定常状態である。例えば、どんなプログラムでも大半の時間は、ループかあるいはイエベント待ちに費やされている。

このような定常状態では、3階以上の変数を考慮する必要はない。しかし、1階の仕様のみを考えるのでは、真の意味での開放分散システムの仕様を考えていることにはならない。ここでは2階の時相論理についての研究を利用しながら、2階のプログラミング方法論について考察する。

2 時相論理で記述するオブジェクト

ここで使う時相論理は区間時相論理(Inteval Temporal Logic またはITL[Mos84])と呼ばれるものである。この論理には以下の3つの様相演算子がある。

chop P& Q 時区間を二つに分け、前半でPが成り立ち、後半でQが成り立つ。

next @P 次の時刻から始まる時区間でPが成り立つ。時区間が終わってしまって、 次の時刻がない 時にはF。

length $length(n)$ 時区間の長さがn。(&と@を使って表すことができる)

ここで、ITLの時間は、0から始まる離散的な時間で、無限に続くが時間の終わりはあるとする。つまり、compact descrite time を仮定する。これにより、無限に続く時区間を特別扱いする必要がなくなる。

筆者らは、ITLを使った論理型プログラミング言語 Tokio [河野 85] について研究をおこなってきた。1階のITLはCなどの普通のプログラム言語を状態や副作用を含めて記述できるくらいに強力であることがわかっている。従って、オブジェクトのメソッドを記述すること自身には問題はない。

しかし、オブジェクトは状態を持っているので、オブジェクトを状態遷移機械とみなした方が簡単である。時相論理の記述は強力であり、極めて複雑なことを短い式で記述することができる。これがプログラミングに向いていると簡単にいうことはできない。プログラミング自身は分かりやすく容易なものであるべきである。この点では、オブジェクトを状態遷移で記述する方がすぐれている。実際、シュライヤー ・メラーの方法では状態遷移をオブジェクトの設計に用いている。

しかし残念ながらオブジェクトの状態遷移は、そのオブジェクトだけで閉じているわけではない。また、複数のオブジェクトがあった場合、単純に状態遷移の積をとるような方法ではシステム全体の状態数は爆発的に大きくなってしまう。また、オブジェクト全体を時相論理式で表すようなことをすると、全体の時相論理式のモデル(その論理式が受けつけるイベントの時間系列の全体)も巨大になってしまう。オブジェクト指向プログラミングは、大きなソフトウェアプロジェクトを対象にしているのであり、これでは本末転倒になってしまう。

シュライヤー メラー法[SM92]などでは、Action Dataflow Diagram (ADFD) という図(図1)を書き、関連する状態遷移を一連の流れの一例として表現する。一例を使うために巨大な状態遷移を図示する必要がなく、プログラムの流れを認識することができる。しかし、ADFDは、どちらかといえば構造化手法に偏った方法であり、システム全体が既に決まった仕様を持っている時に初めて書くことができる図である。

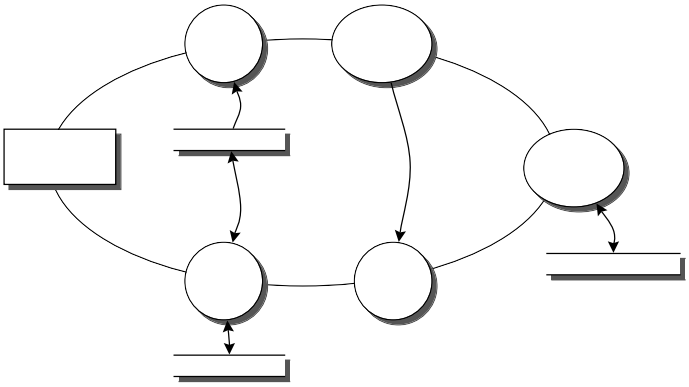

Figure 1: Action Dataflow Diagram

2.1 定常状態にあるオブジェクト

ここでは、メソッド自身は普通のプログラミング言語で記述することにする。これは現実的な対応でもある。ITLは状態遷移図も、その論理要素として含むことができるので、このようにしてもオブジェクトをITLの記述として考えることになんの不都合もない。

ADFDで記述することができるような場合は、関連するオブジェクトは決まっており、その意味で定常状態にある。定常状態ではオブジェクトのやり取りはできないわけだが、ここでは、コンテナクラスのようなデータを構造化しただけのものと、その上の写像からなるものは1階の対象物だと考えて良いことにする。このようにすれば、大半のオブジェクト指向プログラミングのADFDは、2階の論理の範囲内で仕様記述することができる。

しかし、本当のオブジェクト指向プログラムでは、ADFDには常に不定な部分が含まれていることになる。

特定のオブジェクトをプログラムする場合を考えよう。ADFDは、その特定のオブジェクトからみた他のオブジェクトの期待する振る舞いだと考えることができる。他のオブジェクトは、まだプログラムされていなかったり、変更されたりすることがある。この時、通常のオブジェクト指向プログラミングでは、他のオブジェクトに関する記述を、自分のオブジェクトに記述することはできない。

これがオブジェクト指向言語によって表現されたプログラムが読みにくい一つの理由である。プログラムを追っていくとメソッドが飛びまくり簡単に道に迷ってしまう。オブジェクト指向言語によって記述されたプログラムを理解するためには、そのオブジェクト指向設計の時に使われた情報モデルや、状態遷移図、Action Dataflow Diagram (ADFD) を見なくてはならない。これでは、プログラム言語自身の役割としては不満が残る。プログラミング全体をサポートするのがプログラミング言語であるはずで、それが他の図や言語を必要つるならば、それは、そのプログラミング言語にインテグレートされているべきである。

3　2 階の時相論理による ADFD の記述

Action Dataflow Diagram (ADFD) 自身は、プログラムとは異なる。オブジェクト指向言語のプログラムの実現はオブジェクトのメソッドによって行われる。ADFD は、オブジェクトの仕様記述に近い。これは、本来、メソッドと同様にオブジェクトに付随するものでなければならない。

このような仕様記述は、まさに 2 階の ITL によっておこなわれる。そのオブジェクトが期待する、他のオブジェクトの動作の仕様を ITL で記述すれば良い。これは、データや関数のタイプ (型) に対して、メソッドのタイプ (型) だということができる。このようなタイプ (型) としては、最近、π 代数上でグラフタイプというのが提案されている。ITL は基本的には状態遷移に対応するので、状態遷移によりオブジェクトの型を決めることになる。ITL は、状態遷移よりも宣言的に記述することができるので、型宣言としてより優れているということができる。

このような宣言的な同期記述の方法の一つしては、Path Expression というのが提案されている。Path Expression は、論理ではないので否定などがなく記述能力としては低く、また、当然、不定のイベントを含むようなことはできない。ITL は、Path Expression の記述能力をはるかにしのいでおり、決定手続きを持つので Path Expression よりも ADFD の記述に適している。

さらに、並列オブジェクトの場合は、メッセージ送信にさまざまなアノテーションを付けることも良く行われている。ABCL/1[YSTH87] では、現在型、過去型、未来型 (図 2) という 3 種のアノテーションがある。これらは、オブジェクトの自分自身の状態変更という点からは同じ意味であるが、他のオブジェクトのメッセージの待ち合わせという点からすれば、新しいメッセージの待ち合わせという状態を付加していることになる。これらは、明らかにメッセージ間の時相論理的制限を表している。

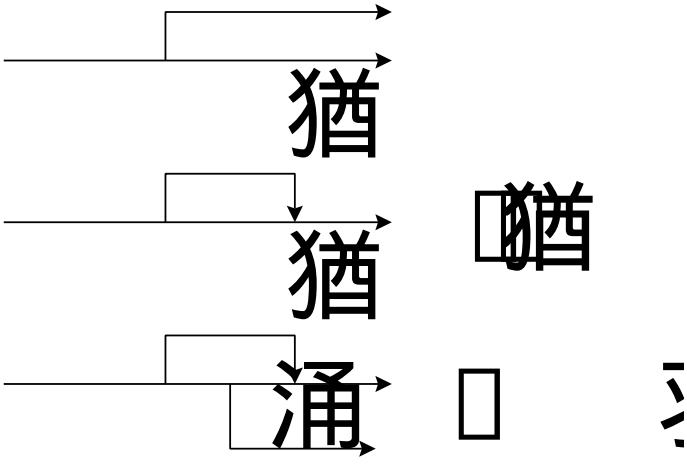

Figure 2: 現在、過去、未来

メッセージ送信を期待する表現を時相論理式を使って、メソッドが構成する状態遷移の中に埋め込むことにより、オブジェクトの能動的なメッセージ送信だけでなく、そのオブジェクトが受動的に期待するメッセージ送受信を記述することができる。これらはオブジェクトが期待する ADFD を構文としてメソッドに埋め込むことに相当する。

例えば、lock などもこのような時相論理によるアノテーションと考えることができる。Unix の System call のほとんどは、(alarm, read, write, select, flock, wait) 時間を含む仕様となっている。これらも、時相論的な記述を同時に含めることによって、プログラミング言語として表現された宣言的なタイミング記述を持つことができる。(図 3) 例えば、これを使ってファイルの close の忘れなどをチェックすることができる。

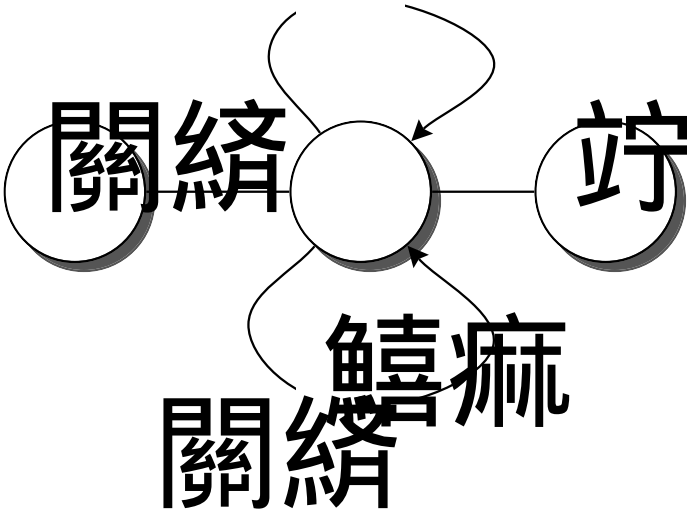

Figure 3: File Interface

さて、これらの記述を使って、どのようなプログラミング技術を実現できるのだろうか? その前に、もう少し 2 階の ITL について考察を行うことにしよう。

3.1　2 階の区間時相論理 2ITL

区間時相論理 ITL は、命題論理の範囲内でも決定手続きはないことが知られている。これはその表現力の高さの裏返しでもある。もし、すべての命題変数の値が時刻にのみ依存し、つまり、時区間の最初にのみ依存し最後には依存しないとすると、著者らによる決定手続き [Kon94] が知られている。この制限は Local と呼ばれる性質であり、Local ITL と呼ばれる論理を構成する。ここでは、さらに closure 演算子 * をつけ加えておく。これにより LITL は、状態列の正規表現 (RE) と同じクラスを表現することができるようになる。

古典命題論理の論理式の値は自由変数の値が決まれば、T/F しかない。従って、高階の命題論理は 1 階の命題論理に帰着する。これは量化記号が入っていても同様のことが成り立つ。2 階の変数がどのような論理式に束縛されても、その値は結局は T/F になる。

古典述語論理でも状況は基本的には同じで、高階の変数はなんからの述語から構成されている論理式に束縛されるが、それを、その式に現れるすべての変数を引き数に持つ新しい述語に置き換えることにより 1 階の述語に変換することができる。しかし、これが実用的な変換だとはいえない。もちろん述語論理には決定手続きは存在しない。

時相論理の場合は、それほど自明ではない。命題論理を考察しよう。実際、LITL の論理式は正規表現 RE に相当す

るが、LITL の変数は一時刻のイベントしか表せない。従って高階 LITL を LITL に帰着させることはできない。

制限のない ITL は、その変数にあらゆる状態列を割り当てることができるので、高階論理と考えることができる。しかし、これでは決定手続きが存在しない。この制限を正規表現まで落としても、残念ながら決定手続きは存在しないことが証明されている。

次の章では決定手続きの存在する ITL のサブセット (で、LITL のスーパーセット) に相当するものを探してみる。

4 自動検証

自動検証にはいくつかの方法がある。良く知られていて実用性が高いのはモデルチェッキングと呼ばれるもので、実装 (例えば有限状態遷移器械) が時相論理式を満たすかどうかを検証する。ここでは、その方法よりも適用範囲の広い直接的な定理証明方法について考察する。ただし、こちらの方が要求される計算量は大きい。

定理証明で用いられる方法の一つは、論理式自身の充足性を調べるものである。充足性とは、その論理式を満たす状態列があることである。完全性のある論理では、ある論理式の否定の充足性がなければ、その論理式は valid であり、定理であることがいえる。LITL は完全性があり、この方法で定理証明をおこなうことができる。

4.1 タブロー法

様相論理の充足性を調べるにはタブロー法 [Gol87] が有効である。この方法では、様相演算子を可能世界に順次展開していく。論理演算子の組み合わせは有限なので、展開により新しい論理式の項が生成されるか、生成されても有限に止まれば、すべての可能な展開を試すことができる。

時相論理の場合は、時刻 0 から初めて順次未来に可能世界を展開していく。展開した論理式が前に出たものと同じならば、その後の展開も同じになるので、それは状態遷移の繰り返しを表す。

この方法は、展開の仕方を決定的 (分岐が排他条件になるようにすること) により、効率的になる。これは、非決定性状態遷移機械を、決定性状態遷移機械に変換する Subset Construction と近いアルゴリズムである。従って、時相論理式の中に状態遷移器械を入れても、まったく同様にタブロー展開することができる。これは、ちょうどモデルチェッキングの拡張になっている。

4.2 モデルの制限

残念ながら ITL や、変数の満たす状態列を正規表現に制限した Regular ITL は、展開時の項の有限性を満たさない。どちらの変数も任意の状態を持つことができるので、次の時刻に展開する際に新しい変数名を使わなくてはならないからである。

$$\begin{aligned} R &= (\mathrm{empty} \wedge R_0) \vee @R_0 \\ R_n &= (\mathrm{empty} \wedge R_n) \vee @R_{n+1} \end{aligned}$$

この $_n$ が、どんどん大きくなることが ITL/Regular ITL の undecidability を表している。

従って、この無限の項の生成を止めればタブロー法は停止する。ここで、タブローの展開に制限を付けて無限の生成を止めることにする。これは、論理式のシンタックスには影響を与えない。実際には、論理式の受け入れるモデルを制限したことになる。実は、Regular ITL もそのようなモデルの制限である。

ここでは以下のような制限を考える。

finite length ITL の制限のもっとも強い制限は、変数の T/F の変化に時間制限を付けることである。これにより、変数の状態数を有限にすることができる。時区間自身は無限長で良いが、ある長さ m を越えるとその値は変わらない。これは以下の式によって表すことができる。

$$\mathrm{limit(P)} \quad (length(m)\&true) \quad \rightarrow \quad (P \quad \equiv \quad ((P \wedge length(m))\&\ T))$$

finite process 次に弱い制限は、変数の表すプロセスの数に制限を付けることである。式の表すプロセスの数は時間によって変化する。このプロセスの数が m を越えると、もっとも古いプロセスの値は T/F に固定されるとする。これは式中に表れる展開された変数の数を制限することに相当する。

finite interraction 最後に、もっとも弱い制限として、変数の相互作用の数に制限を付けることができる。異なる時区間上の 2 階の変数は異なる値を持つ。ある時区間上に単独に現れる 2 階の変数は相互作用しないので、非決定的に T/F になる様相演算子に置き換えることができる。逆に言えば、変数は自分自身としか相互作用しない。相互作用を持つプロセスの数が m を越えると、もっとも古いプロセスの値は T/F に固定されるとする。

これらの制限により、高階の ITL をモデルが制限された条件ではあるが自動検証できることになる。

5 2 階の論理を用いたプログラミング

2 階の論理を用いたプログラミングにはいろいろなものが考えられる。

まず仕様検証として、これらの自動検証を用いて以下のようなことが可能である。

- 定常状態のオブジェクトが指定された時相論理による仕様記述を満たしているかどうかを検証する。
- いくつかのオブジェクトが未定の段階で仕様が矛盾しているかどうかを調べる。
- オブジェクトが ADFD を満たすかどうかを調べる。

もちろん、ここで示した検証方法は計算量がかかるので大きな時相論理式を使うことはできない。プログラム全体を時相論理を使って記述することをせずに、アノテーションとして時相論理を使えば、時相論理式自身は小さくなると期待される。

モデルの制限はアドホックな感じを与えるが、実際のプログラミングでは良く使われる制限であるともいえる。

- タイムアウト
- プロセスの数の制限
- プロセス間の相互作用の制限

ITL の中での制限は、どちらかというと 2 階の同じ変数同士の間で行われるので、これとは意味が違うが、方法論的には同じ制限である。

5.1 オープンエンドなプログラミング

2 階の論理を使うことにより、一部のオブジェクトが不定の段階でも検証できるところが開放型環境での検証を実現している。これは、fairness や eventuality あるいは、schedular などとしても用いることができる。これらは、プログラマとは別に OS や実装によって異なるものであり、いわば無意識のうちに 2 階の変数としてプログラムに組み込まれているものだとも考えられる。

5.2 要求仕様の自動抽出

不定のオブジェクトを含むモデル生成を行うと、それは不定のオブジェクトに対する制約を出力する。これは、一種の逆プログラムであり、不定の部分を含めてプログラムすることにより、不定の部分のプログラミングを進めているいることになる。今は、まだ、2 階の変数に対する制約を出力するようなアルゴリズムはできていないが、出力されたモデルは状態遷移機械を構成するので、その器械を満たすことは必要である。

5.3 既存のプログラムの 2 階化

現在あるプログラムの一部を変数とすれば、それはライブラリとして使えるかもしれない。要求抽出と組み合わせてライブラリの自動生成などができる可能性がある。

5.4 Assertion と Synchronization

時相論理は、ここでは Assertion と Synchonization に使われている。これらは、メソッドを使って実現できるものであるが、オブジェクトの中で明示的に宣言することにより、プログラムの可読性をあげることができる。

Synchonization は、これを実現する状態遷移機械を構成しなくては、実現できない。これはスケジューラオブジェクトを生成するか、OS などの primitive を使って実現することになる。

5.5 プログラミング手法としての制限

モデルの制限は、検証方法による都合であるが、ADFD の形式や Design Pattern などを利用することにより、実質的により小さいモデルを生成することができるかも知れない。プログラムの信頼性は、検証のしやすさにも関係すると考えられる。プログラミング手法の評価の一つとしてモデルの制約をあげることができる。

5.6 リフレクションとの違い

ここでは、高階論理を使ったオブジェクト指向プログラミングを考察したが、同様のものにリフレクションがある。リフレクションは、高階というよりはメタ論理を使った、プログラムの変更や修正を行うものである。リフレクションのモデルは、メタインタプリタを使うものや、OS の kernel をマップしたもの、コンパイル時の部分計算を用いたものなどがある。

ここでの 2 階の論理を用いたプログラミングは、リフレクションのようなメタインタプリタを使うものと異なり、効率に関する問題が少なくてすむ。さらに、メタインタプリタ方式では、システムのあらゆる所にオブジェクトが触れる、あるいは、触れないような工夫を必要とするが、このプログラミング手法ではそういうことはない。

一方で、未定のオブジェクトの部分を適当なオブジェクトに変更することにより、リフレクションと同等の適応性を与えることができる。ただし、それは前もって計算された適応性に限られる。

6 2 階のプログラムの直接実行

生成されたモデルは状態遷移器械なので、これを直接実行することができる。この場合、不定のプロセスは適当に (整合性のある形で) 実行される。論理的に関係ない不定のプロセスは相互作用の制限を使えば、自動的に消滅する。つまり、この時、直接実行は、制限に応じた省略された方法で実行される。

制限されたモデルを生成した後でも、もとの論理式があれば、実行と同時にタブロー法による展開を使えば、それから省略しない詳細な実行を再現することができる。

7 まとめ

以上、2 階の時相論理を使ったプログラミングについての考察をおこなった。実際のオブジェクト指向言語での時相論理仕様の埋め込みや、そのオブジェクト指向設計との関わりを考察した。さらに、高階の時相論理の自動検証や実行についての考察もおこなった。

今後は、検証方法の実用化や、実際のプログラミング言語としての実現などが必要だと考えられる。

References

[Gol87] Robert Goldblatt: Logic of Time and Computation, Vol. 7 of CSLI Lecture Notes, CSLI (1987).

[Kon94] Shinji Kono: A Combination of Clausal and Non Clausal Temporal Logic Program, Executable Modal and Temporal Logics, Vol. LNAI-897, Springer-Verlag (1994), Lecture Notes in Artifificial Intelligence.

[Mos84] B.C. Moszkowski: Executing temporal logic programs, Technical Report 55, Computer Laboratory, Univ. of Cambridge (1984).

[SM92] Sally Shlaer and Stephen J. Mellor eds.: Object Lifecycles, Prentice-Hall (1992).

[YSTH87] A. Yonezawa, E. Shibayama, T. Takada and Y. Honda: Modeling and programming in an object-oriented concurrent language ABCL/1, Object-Oriented Concurrent Programming, MIT Press (1987).

[河野 85] 河野真治，青柳龍也，藤田昌宏，田中英彦: 時相論理型言語 Tokio の実装，Logic Programming Conference '85 (1985)．