ソフトウェア工学 83－16
（1992. 2. 7）

オブジェクト指向プログラム開発時における制約評価

宮永 靖之† 佐藤 康臣† 岡本 康介† 平川 正人‡ 市川 忠男‡

広島大学大学院† 広島大学工学部‡

近年、オブジェクト指向に制約を導入した言語やシステムが数多く開発されている。しかしこれらのシステムにおいて、記述された制約の誤りはプログラムの信頼性を低下させるが、ほとんどのシステムでは制約の誤りはチェックされていない。本研究では、オブジェクトの本質的な性質を表す制約やメソッドの実行条件を表す制約をクラスおよびメソッド定義時に評価し、またテスト時にテストケースを与えて誤った制約を検出することにより、制約間の矛盾を排除するようなプログラム開発方法を提案する。この方法を用いることによって制約の正当性が保証され、信頼性や再利用性の高いソフトウェアの作成が可能となる。

Constraint Evaluation during Constraint Object-Oriented Program Development

Y.Miyanaga, Y.Sato, K.Okamoto, M.Hirakawa, and T.Ichikawa

Faculty of Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 724, Japan

In recent years, there are object-oriented languages and systems which use constraints in order to define the relationships between objects. But in these systems, the correctness of the constraints aren't checked. This paper describes a program development methodology to find the inconsistency among the constraints in an object-oriented constraint system. In the methodology, the constraints in classes and methods are evaluated when the classes and methods are defined. Constraints which can't be evaluated by the system in class definition are evaluated by giving test cases. This methodology helps programmers to make reliable and reusable software.

1 はじめに

近年、ソフトウェアの生産性、信頼性、再利用性を向上させるためにさまざまな研究が行なわれている。その中でもオブジェクト指向言語[1]は、問題の対象を自然に表現することができ、クラスや継承を用いることでモジュール性を高め、既存のソフトウェア部品(クラス)を再利用することができる[2]ため、特に注目されている。

ところが従来のオブジェクト指向言語では、オブジェクト間の関係を記述するにあたっては、それらを宣言的に記述することができないので、メソッドを用いて手続き的に記述する必要があった。しかしこの方法では、オブジェクト間の関係がプログラマには直観的に理解しにくい。そこでオブジェクト間の関係を宣言的に記述する方法として、制約を持ち込む研究がさまざまな分野で行われ[3]-[6]、我々も昨年、オブジェクト間の関係の記述法として制約を導入した制約オブジェクト指向システムISL-xscheme[7]を開発した。ISL-xschemeでは、クラスのインスタンス変数が満たさなければならない条件とメソッドの実行条件、ならびにメソッド実行後にインスタンスが満たさなければならない条件を記述することができる。

制約を導入したオブジェクト指向言語やオブジェクト指向システムでは、オブジェクトが制約を満たしているかどうかをプログラム実行時に評価する。そしてそれによって、制約を満たすように他のオブジェクトの値を変えたり、オブジェクトの値がプログラマの要求を満たしているかを調べる。ここで用いられている全ての制約は、プログラムの仕様を正しく表わしているという仮定のもとで処理が行われる。しかし制約自身の正当性のチェックは、その制約を定義したプログラマに任されており、システムによる制約の正当性のチェックは行われていなかった。そのため、プログラマが予期しない振る舞いをする時には、どの制約が誤っているのかを調べることが困難であった。また制約を導入したことにより、継承を用いてクラスを再利用する時には、クラスの属性やメソッドだけでなく、制約も考慮しなければならず、クラスを再利用することが難しいという問題も生じている。

そこで本研究では、制約を用いてアプリケーションを作成する場合について、それぞれのクラスやメソッドで定義された制約を、それぞれクラス定義ならびにメソッド定義時に評価し、またテスト時にテストケースを与えて誤った制約を修正することにより、制約間に矛盾が生じないようなプログラムを作成する手法を提案する。

以下、2章で制約とオブジェクト指向プログラミングとの関係及び問題点、3章で制約オブジェクト指向システムISL-xschemeの概要、4章で開発時の制約の評価方法について述べた後、5章でISL-xschemeでのプログラム開発時における制約の評価方法を例を用いて述べる。

2 制約とオブジェクト指向

2.1 制約

ここでは、制約に関する用語の定義について述べる。

ある制約c1、c2があり、その制約がとりうる値の領域をD1、D2とする。$D1 \subseteq D2$なる関係が成り立つとき、「制約c1は制約c2を満たしている」、あるいは「制約c1は制約c2より強い」または、「制約c2は制約c1より弱い」という。また$D1 \cap D2 = \phi$なる関係がある時、「2つの制約は矛盾する」という。例えば、c1を$x > 0$である制約、c2を$x > 5$である制約とすると、c1はc2を満たしていないが、c1とc2は矛盾しない。

また制約がプログラムの仕様を正確に表わしていて、データが制約を満たしていれば必ずエラーを起こすことなく実行できるとき、その制約はクラスあるいはメソッドに対して正当性をもつという。

2.2 制約とオブジェクト指向プログラミング

オブジェクト指向プログラムにおいては、クラスにインスタンス変数を定義することでhas-a関係を、またクラス間に継承関係を定義することでis-a関係を定義することができる。しかし、これらの関係はオブジェクト(クラス)の構造的な関係であり、あるクラスのインスタンス変数aとインスタンス変数bが等しいなどといった意味的な関係を記述することは、宣言的にはできなかった。そのため先のような意味的な関係を定義するには、変数aあるいはbの値が変わるときに、互いにもう一方の値を変更するためのメッセージを送るといった手続き的な記述をメソッド中に埋め込む必要があった。この方法では、インスタンス変数aとインスタンス変数bが等しいことはプログラムを見ただけでは理解しにくい。

そこで構造的な記述ができるオブジェクト指向言語に、意味的な関係を宣言的に記述できる制約を導入したオブジェクト指向言語やオブジェクト指向システムが開発されている。ThingLabII[3]やGarnet[4]はユーザインタフェース構築用のシステムであり、グラフィカルオブジェクトの関係を定義することにより、あるウィンドウの大きさや位置を変えると制約に従って他のウィンドウの位置や大きさが変わる。一般のプログラムの開発に向けての試みとしては、Kaleidoscope、Eiffel、ISL-xschemeなどがある。Kaleidoscope[5]では、メソッド内においても宣言的な記述ができるため、プログラムの記述を削減することができる。またEiffel[6]では、宣言的なassertionを記述することによって、プログラムの正確さと頑丈さを向上させている。著者らが以前提案したISL-xschemeでは、インスタンス変数間の関係、インスタンスが満たさなければならない条件、メソッドの実行に関する条件を制約として記述できる。

このように制約を導入したオブジェクト指向言語やオブジェクト指向システムの利点としては、

- オブジェクト間の関係を宣言的に記述できるため、プログラムを理解するのが容易である。
- オブジェクト間の関係はシステムにより保たれるため、プログラマはオブジェクト間の関係を宣言的に記述するだけでよく、関係を保持するための処理については一切記述する必要がない。

ということが挙げられる。

2.3 制約の導入により生じる問題点

制約を導入したオブジェクト指向言語やオブジェクト指向システムは、記述された制約を満たすようにシステムの状態が適宜システムによって変更される。そのため、プログラマが誤った制約を定義した場合には、プログラマの予期しない振る舞いをする。しかもこのことがプログラマには分からないまま実行されてしまうことがありうる。たとえプログラムが誤った振る舞いをしていることにプログラマが気付いたとしても、制約に沿った値の変更の影響が伝播するため、どの制約が誤っているのかを見つけることが難しい。

さらに、これらの制約はすべてプログラマが定義するため、メソッドの実行条件のように、実際にメソッドを実行するために必要な条件とプログラマが定義した制約条件が一致しない場合がある。その際には、実際にはメソッドの実行条件を満たしているのにプログラマが定義した条件に対してエラーになったり、反対にメソッドの実行条件を満たしていないのにメソッドを実行し、誤った結果を返すということが起こり、ソフトウェアの信頼性を低下させる原因になる。

加えて、制約を導入した場合には、プログラマが継承を用いて再利用したいと思うクラスがあっても制約によって利用できない場合がある。例えば、クラスライブラリに大きさが10であるスタッククラスが定義されている。この時にプログラマが大きさ100のスタックを必要とする場合には、定義されているスタックを再利用することができない。また、プログラマが定義した制約間で矛盾が生じる場合もある。

3 制約オブジェクト指向システム ISL-xscheme

我々が開発した制約オブジェクト指向システム ISL-xscheme はオブジェクト間の関係を制約として記述できる。また、関連するオブジェクトとの関係から自らの状態や振る舞いを変えることのできるジェネリックオブジェクトの導入により、プログラマの抽象的な要求をそのままプログラミングに反映できるといった特長がある。

3.1 制約の種類

ISL-xscheme で扱う制約には class invariant 、precondition 、postcondition の3つがあり、それぞれ以下のような制約を表す。

- class invariant

  クラスのインスタンス変数が実行中に常に満たさなければならない条件を表わす。これはクラスの本質的な性質を意味する。class invariant はすべてのインスタンスに適用され、下位クラスにも継承される。

- precondition

  メソッドの起動にあたって満たされるべき条件を表わす。precondition が満たされた状態でメソッドが実行された時には、postcondition と class invariant の成立が保証されるものとする。

- postcondition

  メソッドを実行した後、インスタンスが満たすべき条件を表わす。

制約は図1のようなシンタックスで記述する。

```
<Constraints>::=({or} <Constraint> {<Constraint>}...)
<Constraint> ::=(<Relation> <Expression> <Expression>)
               | (name! constraint-name <Constraint>)
<Expression> ::=(<Operation> <Expression> <Expression>)
               | instance-variable | constraint-name
<Relation>   ::= < | <= | > | >= | = | >> | type?
<Operation>  ::= + | - | * | /
```

図 1: 制約のシンタックス

3.2 クラス定義

ISL-xscheme においてクラスの新規作成は、図2(a) のようにクラス名、インスタンス変数、クラス変数、スーパークラスを添えて、class クラスに"new"メッセージを送ることによって行なわれる。

```
(class 'new 'PositiveNumber
  '(value)
  '()
  object)
```

(a)

```
(PositiveNumber 'answer 'sub '(arg)
  '((set! value (- value arg))      ;method-body
  '((>= value arg))                 ;precondition
  '((= value (- (old 'value) arg)));postcondition
)
```

(b)

```
(PositiveNumber 'definvariant!
  '((>= value 0)))
```

(c)

図 2: クラス定義例

またメソッドの定義は、定義したいクラスに"answer"メッセージを送ることによって行う(図2(b))。その際、メソッド名とメッセージ引数ならびに、それぞれのメソッド本体、メソッドの precondition 、postcondition を記述する。precondition 、postcondition では、メソッド引数や他のオブジェクトのインスタンス変数の値を参照できるほか、メソッドの実行前の値を old という特別のオブジェクトを用いて参照することができる。

クラスが本質的に満たさなければならない制約である class invariant は、クラスに"definvariant!"メッセージを送ることによって定義できる。

3.3 制約の評価

それぞれの制約は、オブジェクトにメッセージが送られてから評価される。オブジェクトにメッセージが送られると、実行前に class invariant とメソッドの precondition をチェックする。正しければメソッドが実行され、実行後に postcondition と class invariant をチェックする。class invariant のチェックで違反が発見された場合には、制約を満たすように変数の値を変更する。それでも class invariant が満たされない場合にはユーザに警告を出して、ユーザの指示によって適切な例外処理を行なう。

3.4 制約間の関係

ISL-xscheme では、class invariant 、precondition 、postcondition の間に、次のような関係が成り立っている必要がある。

1. 下位クラスの class invariant は、上位クラスの class invariant より強くなければならない。

   class invariant は上位クラスの class invariant を継承する。そのためあるクラスの class invariant が上位クラスの class invariant より弱い場合、自分自身のクラスに定義されている class invariant を満たしていても上位クラスの class invariant は満たさないという場合が起こり得る。例えば、A というクラスがあり、その下位クラスとして B というクラスが定義されているとする。A には class invariant として $x > 5$ であるという制約が、B には $x > 0$ であるという制約が定義されている時、B クラスのインスタンス b において x の値が 2 であるとすると、B の class invariant は満たしているが A の class invariant は満たさないということが生じ、B の class invariant である $x > 0$ は意味をなさなくなる。このことは、クラス B を再利用するときに、B の class invariant に基づいてメソッドを作成する場合に問題となる。また、上位クラスと下位クラスの class invariant に矛盾がある場合には、両方のクラスの class invariant を満たす値が存在しないということが生じる。

2. precondition および postcondition は class invariant を満たさなければならない。

   メソッドの実行は、インスタンスが class invariant を満たしている状態で行われるので、class invariant より弱い precondition が定義されている時には、precondition に誤った制約が記述されていることを意味する。例えば、class invariant に $x > 0$ 、メソッドの precondition に $x \geq 0$ という制約があった場合、インスタンス変数 $x = 0$ となる状態ではメソッドが実行されないので無意味である。このような意味のない情報はクラスの再利用の際にプログラマによけいな負担を生じさせる。また両制約が矛盾している場合には、該当するメソッドは常に実行不可能である。

   postcondition が class invariant を満たしていない場合にはエラーが起こり得る。これはメソッドの記述あるいは定義された制約が誤っていることになり、変更する必要がある。例えば、postcondition が $x > 4$ として class invariant が $x > 5$ の場合には、メソッドの実行後結果として $x = 5$ となったとすると class invariant が満たされなくなるのでエラーが生じる。

プログラマは新しくクラスを作成する場合、以上のことを考慮しながら制約を記述しなければならず、プログラマの負担が増大される。

3.5 制約の優先度

ある状況においては、必ずしもすべての制約は満たさない時がある。この場合には通常はエラーとなるが、ある状況ではどちらかの制約を満たせばよいという場合もある。例えば、ある製品を作成するにあたって、かかる費用は所定の金額以下に押さえなければならないが、納期は少し遅れてもよいという場合がある。

制約の優先度を定義する方法としては次の 2 つがある。まずプログラマが制約間で優先させたい制約があれば class invariant の定義時にそれを明示する。例えば、制約 c1 、c2 があり、c1 を c2 より優先させたい場合、class invariant 定義時に、"c1 ≫ c2"という優先順序に関する制約を付加することにより定義する。もう一つの方法としては、テスト時に優先順序を定義する。制約間に矛盾がないか否かは定義時にはわからない場合があり、あるオブジェクトの値によってはテスト時に矛盾が生じ得る。システムから矛盾する制約が発生したというメッセージを受けて、プログラマがその時点で優先して満たしてほしい制約を指定することにより、次からは優先順位が高いものから満たされる。

4 開発時の制約評価

通常のオブジェクト指向言語でプログラムを作成する場合、プログラマは、必要となるクラスの属性ならびにインタ

フェースから、作成しようとしているクラスの上位クラスを決定する。その後、クラスで必要とするインスタンス変数、クラス変数を定義し、そのクラスで必要とするメソッドを定義する。制約オブジェクト指向言語では、これらに加えてオブジェクト間の関係を制約により定義することができる。しかしながら、2.3、3.4で述べたように、定義する制約の間に矛盾がないか等をユーザ自身で調べなければならない。本研究では、開発時にConstraintクラスを用いて制約の評価を行うことで、このような問題に対処する。開発時の制約処理は、与えられた制約に対して充足できる場合には値を求め、充足しない場合には式を簡略化する。これによりテスト時のプログラマの負担を軽減できる。プログラム開発時にどのように制約が評価されるのかをclass invariant、precondition、postconditionについてそれぞれ述べる。

4.1 制約の評価方法

クラスおよびメソッドで定義された制約は、以下の手順で評価される。

手順1: それぞれの定義時にConstraintクラスを用いてチェック

ここでは、今までに定義されている制約間の依存関係に一貫性があるかどうかを調べる。この処理を司どる特別なクラスとしてConstraintを用意する(詳細は後述)。制約処理能力は、あとで述べるsatisfy?、checkメソッドに依存し、これらのメソッドの能力を上げることによって制約処理能力を上げることができる。

手順2: テストケースで実際にデータを与えて制約をチェック

Constraintクラスを用いて評価できなかった制約や違反しているかどうかを判断できなかった制約について、実際にテストデータを与えて制約を評価する。

これによって評価された制約は、正当性をもつ。

4.2 Constraintクラス

制約を開発時に評価するためにConstraint、Constraint-Spaceクラスを導入する。プログラマがそれぞれのクラスやメソッドで定義した制約は、Constraintクラスのインスタンスとして Constraint-Spaceクラスのインスタンス変数constraintsに保持される。Constraintクラス、Constraint-Spaceクラスは図3のように定義される。

Constraintクラスには、インスタンス変数としてname、var、expression、preference、そしてメソッドとしてadd、check、satisfy?、simplify、show、new、create-precondition、create-postconditionが定義される。以下では、それぞれについて説明する。

name 制約が定義されているクラスとその制約がclass invariantであるか、あるいはメソッドのprecondition、postconditionであるかを区別するために使われる。

```
(class 'new 'Constraint
       '(name var expression preference) '() object)
(<constraint> 'add exp)
(<constraint> 'check exp)
(<constraint> 'satisfy? exp)
(<constraint> 'simplify)
(<constraint> 'show-constraint)
(Constraint 'new
         class-name constraint-type exp)
(Constraint 'create-precondition
         class-name method-name)
(Constraint 'create-postcondition
         class-name method-name)

(class 'new 'Constraint-Space
       '(constraints) '() object)
(Constraint-Space-Object 'add <constraint>)
(Constraint-Space-Object 'delete
       class-name constraint-type {method-name})
(Constraint-Space-Object 'take-constraint
       class-name constraint-name {method-name})
```

図3: 制約処理のためのクラスの定義

var 定義された制約式で使われている変数を(変数名 簡略化した式)というリストで保持している。また簡略化した式には、その変数がとりうる最小値と最大値が保持される。

expression 定義された式をそのままの形で保持している。制約式には、他の制約と区別するための制約名がつけられる。この制約名は、プログラマが制約の優先順位をつけたい時などに使用する。

preference プログラマが定義した制約の優先順位を((制約1 制約2)(制約3 制約4 ...)(...)) という形で保持する。この場合、制約1と制約2が最も優先順位が高く、次に制約3と制約4の順位が高いと解釈する。

add expressionに制約を追加する。追加される制約は既にある制約との間に矛盾がないものに限られる。

satisfy? 与えられた制約がexpressionにある制約を満たしているかどうかを調べる。

check 与えられた制約がexpressionにある制約と矛盾がないかどうかを調べる。

simplify expressionにある制約を簡略化する。

show-constraint 簡略化した式を表示するためのメソッドである。

add-preference プログラマが定義した制約間の優先順位に矛盾がないかどうかを調べ、矛盾がなければ制約間に優先度をつける。具体的には、与えられた優先順序が矛盾しないかをインスタンス変数preferenceに保持されている優先リストを用いてチェックする。矛盾が生じ

る場合にはプログラマに知らせ、矛盾がない場合には preference に制約を付加していく。

create-precondition 、create-postcondition メソッドが定義された時に、そのメソッドを実行するために最低限必要な制約をメソッド本体から導出する。precondition は、メソッドがエラーを起こさないための条件を表わしているので、メソッド中のインスタンス変数にメッセージを送る場合に、そのメッセージの precondition をインスタンス変数が満たしていなければならない。また、postcondition は、インスタンス変数に値を代入する set! 文から取り出すことができる。

制約式とそれに対応する Constraint クラスのオブジェクトの例を図4に示す。

```
(Stack 'definvariant!
   '((>= element-num 0)
     (>= max-size element-num)
     (= max-size 10))
```

name	Stack invariant		
var	element-num	0	10
	max-size	10	10
expression	(>= element-num 0)		
	(>= max-size element-num)		
	(= max-size 10)		
preference	()		

図 4: 制約と制約インスタンス例

Constraint-Space クラスはシステム中にただ一つのインスタンスである Constraint-Space-Object を生成する。Constraint-Space-Object は、すべてのクラスの class invariant とメソッドに関する precondition 、postcondition をインスタンス変数 constraints に保持している。また、Constraint-Space のメソッドとして add 、delete 、take-constraint を持っている。add メソッドは Constraint クラスから生成されたインスタンスをインスタンス変数 constraints に追加する。delete メソッドは constraints にある制約インスタンスを削除する。また take-constraint メソッドは、指定されたクラスの制約を constraints から探す。

4.3 class invariant の評価

class invariant は、クラスに"definvariant!"メッセージを送ることによって定義される。class invariant が上位クラスの class invariant を満たしているか否か、また定義した class invariant の間に矛盾がないかは以下のようにしてチェックされる。

1. 注目しているクラスの class invariant がすでに定義されているかどうかを Constraint-Space-Object の中から探し、もしあれば check メッセージを送り、定義される class invariant が矛盾を生じないかどうかを調べる。Constraint-Space-Object にそのクラスの class invariant がなければ Constraint クラスに new メッセージを送り、新しく制約インスタンスを生成する。

2. 制約インスタンスが上位クラスの class invariant を満たしているかを調べるために、satisfy?メッセージを上位クラスの制約インスタンスに送る。もし、上位クラスの class invariant を満たしていなければ、どの class invariant が上位クラスのどの class invariant を満たさないのかをプログラマに知らせる。ここでプログラマは、違反が生じた制約を変更するかどうかを決定する。制約を変更する場合には、上位クラスの class invariant を満たすように変更する。制約を変更しない場合には、システムはその上位クラスに新たな上位クラスを作成し、矛盾する制約を除いた属性をその共通の上位クラスに移動する。その後、作成された新たなクラスを定義したいクラスの上位クラスとする。これによって、上位クラスと下位クラスとの関係を保ちながら、プログラマが望む class invariant を定義することができる。

3. 上位クラスの class invariant を満たしていれば、その class invariant を制約インスタンスに追加する。class invariant 間に矛盾があった場合には、矛盾する class invariant をプログラマに示す。

4.4 メソッドの定義

メソッドを定義する時に、そのメソッドの precondition 、postcondition は以下のようにして調べられる。

1. メソッドの実行条件を求める create-precondition メッセージを Constraint クラスに送る。

2. 上位クラスに同名のメソッドが定義されているかどうかを調べ、上位クラスにそのようなメソッドが定義されていれば、そのクラスの precondition を満たしているかを調べる。

3. 与えられた precondition が class invariant を満たしているかどうかを調べる。満たしていなければ、該当する precondition を削除する。

4. プログラマが定義した precondition が、create-precondition メッセージにより求められた条件を満たしているかどうか調べる。もし満たさなければプログラマに知らせ、必要な場合には precondition を修正する。

postcondition についても同様に調べ、必要に応じて矛盾が生じないように修正する。

4.5 テストケース実行時の処理

テストでは、プログラマが作成したクラスやメソッドに実際にデータを与えてメソッドを実行させることにより、作成したクラスの正しさの検証およびバグの検出を行う。本手法では、それらに加えて、定義した class invariant 、precondition 、postcondition の正当性の検証、および定義した制約間に矛盾が生じた場合の制約の変更の支援や制約の優先度の設定を行う。テスト時の制約の処理手順を図5に示す。

テストではまず、定義したクラスに対して新しくインスタンスを生成する。そしてメソッドの実行に必要なオブジェクトがあれば、それらのインスタンスも生成する。次に、生成したインスタンスに対して、メソッドを実行する。このとき、3.3で述べた方法とは違い、そのインスタンス、メソッドに関するすべての制約が調べられる。まず、定義したクラスの class invariant を満たしているかどうかを調べる。満たしていない場合には、与えられた値を満たすように class invariant を変更する。この後に、上位クラスの class invariant を満たしているかどうかを調べる。ここで満たしていない場合には、3.4で述べた関係を満たしていないことになるので、違反を生じた制約を除いたすべての class invariant 、インスタンス変数、メソッドを持つクラスを生成し、そのクラスを定義したいクラスの上位クラスとする。これによりクラス間の関係を保つことができる。さらに、precondition を満たしているかどうかを調べる。もし満たしていない制約が create-precondition で生成された制約であれば、定義したメソッドは実行不可能であるので、必要に応じてメソッドを変更する。もしそうでなければ precondition を緩める。

そして次に、上位クラスの precondition を満たしているかどうか調べる。上位クラスの precondition を満たしていない場合には、そのメソッド名を変更することにより、クラス間の一貫性を保つ。

その後メソッドを実行し、postcondition を満たしているかどうかを調べる。postcondition を満たしていない場合にはメソッド本体か postcondition を変更する。そして最後に class invariant 、上位クラスの class invariant を調べる。

以上のことをすべてのテストケースに対して行うことにより、クラスと制約はプログラマの要求を満たしているものと考えることができる。

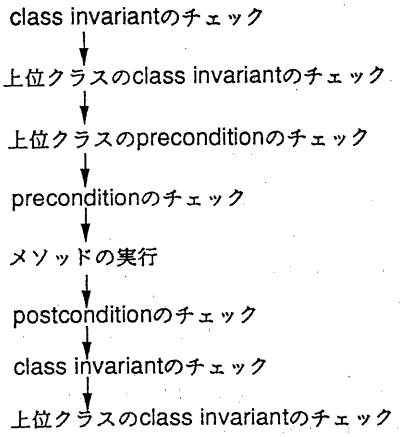

図 5: テスト時の制約処理

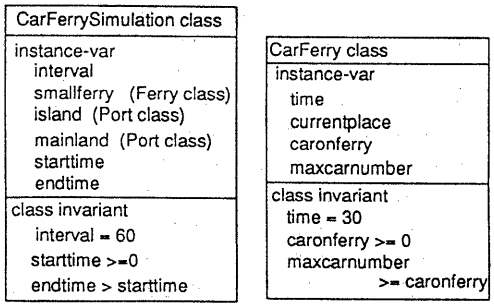

図 6: フェリーシミュレーション

5 プログラム開発例

5.1 フェリーシミュレーション

ここでは例としてシミュレーションについて考える。これは、ある島と本土を運航するフェリーのシミュレーションである。すでに、クラスとして CarFerrySimulation 、CarFerry 、Port 、Car 、Exponential というクラスが定義されている(図6)。

CarFerrySimulation クラスは、シミュレーションの初期設定から実行の制御までを行うクラスである。このクラスにはインスタンス変数としてフェリーの運航時間を設定する interval とシミュレーションを構成する smallferry 、island 、mainland 、またシミュレーションを行う時間を指定する starttime 、endtime が定義されている。このクラスの class invariant には、"interval = 60"という制約が定義されている。また CarFerry クラスには、フェリーの所用時間を表す time 、現在のフェリーの場所を示す currentplace 、乗船している車を示す caronferry 、ならびに最大乗船車数を示す maxcarnumber がインスタンス変数として定義されている。また CarFerry クラスの class invariant として"time = 30"が定義されている。Port クラスには、インスタンス変数として地名を表す place と、フェリーを待っている行列を表す waitcar が定義されている。Exponential クラスは、指数分布に従うデータを出力するクラスで、新しくシミュレーションで用いる車データを生成する時に使用される。

5.2 開発例

ここでは5.1で述べたクラスを再利用して、以下の機能を追加する作業について考える。

1. フェリーの運航時間を自由に変更できるようにしたい。
2. フェリーの所用時間は20分である。
3. フェリーに人も扱うようにしたい。

まず1.の要求に対しては、CarFerrySimulationクラスのメソッドとして"set-interval!"を新しく定義し、このメソッドを実行することによってフェリーの運航時間を自由に変更できるようになる。次に2.および3.の要件から、新しくCarFerryクラスのサブクラスとしてIslandFerryクラスを作成し、インスタンス変数として乗客数を表すpersonnumberと最大乗客数を表すmaxpersonを追加する。またclass invariantとして新たに"time = 20, 0 ≤ personnumber ≤ maxperson"を追加する。このclass invariantは、"(IslandFerry 'definvariant! '((= time 20)(>= personnumber 0)(>= maxperson personnumber))"を実行することにより定義される。この時にclass invariantは、次のように評価される。

IslandFerryクラスのclass invariantはまだ定義されていないので、"time = 20"という制約を定義する新しい制約インスタンスを生成する。そして上位クラスのCarFerryクラスのclass invariantを満たしているかどうかを調べる。しかしCarFerryクラスのclass invariantである"time = 30"を満たしていないのでプログラマに知らせる。この場合、プログラマはtimeを20に設定したいので、そのことをシステムに知らせる。この操作に対応してシステムはCarFerryクラスと定義したIslandFerryクラスの上位クラスとして新しくクラスを生成する。そして矛盾した制約を除いたすべてのclass invariantとインスタンス変数、メソッドを移動させる。

次に4.5で述べたように、新しく作成したクラスとメソッドについてテストを行う。CarFerrySimulationクラスに新しくsamplesimulationというインスタンスを生成する。ここでsamplesimulationに"(samplesimulation 'set-interval! 90)"というメッセージを送ったと仮定すると、この要求はメソッド実行後のclass invariantである"interval = 60"を満たしていないので、その旨の情報がプログラマに伝えられる。プログラマには、このCarFerrySimulationクラスのclass invariantは不必要だとすると、システムはCarFerrySimulationクラスの上位クラスとして新しく別のクラスを生成し、そこにCarFerrySimulationクラスのインスタンス変数、メソッドを移動させる。これによってプログラマはクラス間の制約の関係を考慮する必要もなく、クラスを再利用してソフトウェアを作成することができる。

6 まとめ

本稿では、クラス定義時ならびにメソッド定義時に、クラスやメソッドで定義された制約を評価し、またテスト時にテストケースを与えて誤った制約を検出することにより、制約間の矛盾を排除するようなプログラム開発手法について提案した。プログラム実行時だけでなく、開発時にも制約を評価することにより、誤った制約を容易に見つけることができる。このためテスト時のプログラマの負担を軽減できる。また制約間の矛盾等の検出を支援することにより、プログラムの信頼性も向上する。

今後の課題として、

1. 実行時の矛盾が生じた時の処理、
2. 制約記述能力、
3. 大規模なプログラム開発での制約評価の正しさの検証

が挙げられる。

参考文献

[1] A. Goldberg and D. Robson, "Smalltalk-80 – The Language and It's Implementation – ", *Addison - Wesley*, 1983.

[2] G. Booch, "Object-Oriented Development", *IEEE Trans. on Software Engineering*, pp.211–pp.221, Feb., 1986.

[3] J. H. Maloney, A. Boring and B. N. Freeman - Benson, "Constraint Technology for User-Interface Construction in ThingLab II", *ACM Proc. of OOPSLA'89*, pp.381–pp.388, Oct., 1989.

[4] B. A. Myers, D. A. Guise, R. B. Dannenberg, B. V. Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal, "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces", *IEEE COMPUTER*, vol.23, No.11, pp.71–pp.85, Nov., 1990.

[5] B. N. Freeman-Benson, "Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming", *ACM Proc. of OOPSLA'90*, pp.77–pp.88, Oct., 1990.

[6] B. Meyer, "Object-Oriented Software Construction", *Prentice–Hall*, 1988.

[7] 森本、佐藤、岡本、宮永、田中、市川, "オブジェクト指向環境におけるジェネリックオブジェクト", 情報処理学会ソフトウェア工学研究会、Feb., 1991.