修 士 論 文

アスペクト指向言語による設計制約確認方式に関する研究

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

寺島 真介

2006 年 3 月

修 士 論 文

アスペクト指向言語による設計制約確認方式に関する研究

指導教官　岸 知二 客員教授

審査委員主査　岸 知二 客員教授
審査委員　片山 卓也 教授
審査委員　鈴木 正人 助教授

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

410084 寺島 真介

提出年月: 2006 年 2 月

概 要

近年組み込みシステムの開発は大規模，複雑化しており，開発手法も大きく変化している．これに伴い，組み込みシステムにおいても分析，設計工程が重要であるとされ，今まではあまり用いられてこなかったオブジェクト指向のアプローチが有力視されている．オブジェクト指向の開発法は大きく，分析，設計，実装の 3 つに分かれている．そのオブジェクト指向の開発法を組み込みシステムの開発に適用することで，分析，設計工程において組み込みシステムを考慮した設計が可能になる．しかし実装工程において，設計モデル作成時に考慮された，設計制約が全て反映されているか確認する必要があるが，容易に確認する方法やツールが少なく，非常に手間のかかる作業となる．そこで本研究ではオブジェクト指向の発展に伴い注目されつつあるアスペクト指向を用い，設計制約が実装されたプログラムに反映されているかを容易に確認することの出来る仕組みを提案することと，その機能を実装したツールを作成することの 2 項目を目的とする．

本論文では，まず，設計制約に関して整理し，設計制約とアスペクトの関係にまとめた．はじめに設計制約が決められる設計工程を，アーキテクチャ設計工程と詳細設計工程に分けて考える．分けられた設計工程の特徴と，その工程により決められる設計制約に関して整理した．整理した設計制約から，アスペクトとして捉えることのできる横断的な関心事との関係をまとめた．次に，設計制約の確認対象とした，設計制約の表現方法と確認方法について定義した．最後に提案する確認方法を実装したツールを用いて，容易に設計制約を確認することが出来るか評価を行った．

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景

組み込みシステムが大規模，複雑化する現在，その開発方法も大きく変化してきている．これまでの組み込みシステムは小規模なものが多く，分析工程，設計工程を重視しないプログラミング中心の開発方法が用いられてきた．しかし，開発規模が大きくなるにしたがい，分析・設計工程を重視しない開発方法による問題点が多く指摘されている．そこでこれらの問題に対し，組み込みシステムにの開発にオブジェクト指向を導入した開発方法が有力視されている．

オブジェクト指向の開発方法は，大規模なシステムのモデリングやソフトウェアの再利用性などに優れた実績を持ち，広く普及している．開発方法は大きく分析，設計，実装の3工程にまとめられる．

はじめに，システムの論理構造を把握して，分析モデルを作成する分析工程がある．次に，分析工程によって作成した分析モデルを具体的に実現するための設計モデルを作成する設計工程がある．最後に設計工程によって作成した設計モデルを元に実装を行う実装工程がある．

組み込みシステムには他のシステムに比べ，時間制御や資源の取り扱いなどに厳しい制限が存在する．一般に分析工程においては，論理構造に注目するため並行処理などなどの実現方法や，それにまつわる制約事項を考慮せずにモデル化を行う．設計工程では，それを踏まえ，制約事項を考慮しつつ，具体的な実現方法を検討する．しかし，実装工程では，こうした設計上の制約や判断を踏まえてなされるので，それが正しく反映されているかを確認することが必要となるが，それら確認を容易に行う方法やツールは少ない．しかし一方でオブジェクト指向を開発方法に適用しても，解決できない問題もある．そこでこれら問題に対してはアスペクト指向を用いる開発方法が多く提案されている．

アスペクト指向とは

本研究ではアスペクト指向に焦点を当てる．アスペクト指向とは，複数のオブジェクトに存在している共通の機能を，横断的関心事と呼び，その横断的関心事をアスペクトとして一つにまとめ，独立させることで保守性，再利用性を向上させる考え方である．

本研究ではアスペクト指向言語として，AspectC++を使用することとする．これは今回の研究対象としている事例がC 言語が用いられていること，組み込みソフトウェアにおいては依然としてリソースを消費しない C/C++[1] が用いられていることなどが挙げ

られる．図 1.1 にアスペクト指向言語の使用例を挙げる．ここでは各クラスに存在するロ

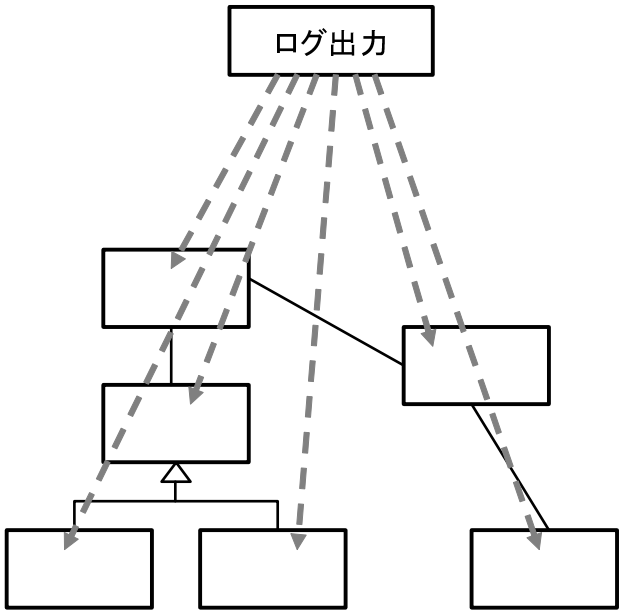


図 1.1: アスペクト指向言語利用イメージ

グ出力機能を横断的関心事と捉え，一つのアスペクトとして取り扱う例である．このように複数のクラスに横断的に存在しているログ出力機能をアスペクトとしてまとめ，ログを出力してほしい部分に直接記述することなく，アスペクトとした機能を追加することができる言語である．アスペクト指向言語を用いることで，プログラム上に冗長な記述などが少なくなり，可読性の向上などの効果がある．

本研究で用いる AspectC++は，Java の代表的なアスペクト言語である AspectJ をもとに作られており，アスペクトは JoinPoint，PointCut，Advice などで構成される．AspectC++で記述されたアスペクトを，対象のプログラムに織り込むことをウィーブ (weave) と呼ぶ．AspectC++のサポートする機能としては AspectJ よりも若干少ないが，使用方法など大きな違いは無い言語体系となっている．

アスペクト指向を開発方法に適用し，設計制約を横断的な関心事とすることで，設計制約を容易に確認することが出来るようになることが期待される．アスペクト指向を用いる利点としては，設計制約を確認するための仕組みを，プログラム中に直接記述する必要がなくなること，またプログラム中に直接記述しないために，アスペクトを外すことで，通常のシステムに戻すことができるなどが挙げられる．また，確認する設計制約ごとに，アスペクトで記述される設計制約を確認するための仕組みを切り替えることで，様々な設計制約を容易に確認することができるなどの利点がある．

JoinPoint
実行時にアドバイスの実行を織り込むことが可能なプログラム上の位置を示す．織り込む位置としては，任意の位置ではなく，メソッドやコンストラクタの実行などの，プログラム作成者にとって意味のある位置となる．

PointCut
条件を指定することで，AspectC++プログラム内に存在するすべての JoinPoint の集合から抽出される部分集合である．

Advice
対象とするプログラムの実行が，JoinPoint に差し掛かったときに，実行される機能のことである．実行のタイミングとしては，JoinPoint の事前 (before 句)，事後 (after 句)，呼び出し時に変わりに実行 (around 句) がある．これら実行されるタイミングについては図 1.2 に示す．

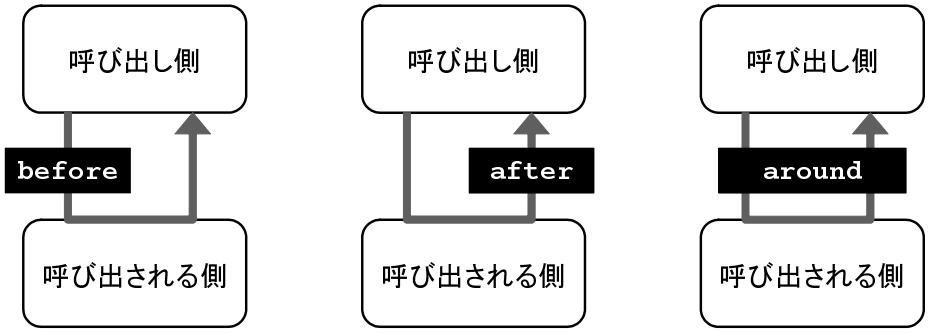


図 1.2: アドバイスの呼び出されるタイミング

その他，対象としている JoinPoint に情報を持った API として ThisJoinPoint(tjp) がある．JoinPoint API を用いることで，JoinPoint の引数の数や，引数，返り値など様々な情報を取得することができる．

1.2 目的

ソフトウェアの開発では，設計制約が，実装に反映されていることを確認する必要がある．しかし容易に確認する方法，ツールなどが少ないという問題がある．そこで本研究では，アスペクト指向を用いて，設計制約が反映されているかを，プログラム上で確認する仕組みを提案すること，その仕組みを用いて確認できるツールを作成することの 2 点を目的とする．

1.3　本論文の構成

本論文の構成は以下の通りである．第 2 章では対象とする設計制約とアスペクト指向の関係などについて述べる．第 3 章ではアスペクト指向を用いた設計制約確認方法について，アスペクトとして捉えることの出来る設計制約の，具体的な表現方法と確認方法について述べる．第 4 章では，設計制約確認方法を実装したツールに関して，システム構成，機能などについて詳細に述べる．第 5 章にて 4 章に説明する設計制約確認ツールを用いた事例の評価について述べ, 第 6 章にて考察を行い本論文を総括する．

第2章　設計制約について

2.1　概要

設計制約とは，設計者が設計時に作成する，必ず満たすべき設計の意図のことである．この設計制約は設計工程にて定義される．本章ではこれら設計制約が定義される設計工程を，アーキテクチャ設計と詳細設計に分けて考え，設計制約について整理を行う．

アーキテクチャ設計は，分析工程によって定義された要求仕様を，効率よく実行するための構造や，再利用性，保守性を考慮してアーキテクチャ設計モデルを作成する工程である．アーキテクチャ設計では実装プログラミング言語に大きく依存しない範囲の設計を行う．例として，この工程で全体の構造や，流れなどを決める．

詳細設計は，実装するプログラミング言語を考慮し，アーキテクチャ設計において決められた構造を考慮した詳細設計モデルを作成する．一般にこの詳細設計工程で決められる設計制約は，アーキテクチャ設計工程で決められる設計制約とは異なる．そこで設計制約を図 2.1 に示したそれぞれの設計工程により分類する．

2.2　設計制約の分類

以下，二つの設計工程において，それぞれ定義される典型的な設計制約について説明をする．

2.2.1　アーキテクチャ設計

アーキテクチャ設計は，要求仕様を効率的に実行するための構造や，再利用性，保守性を考慮したアーキテクチャ設計モデルを作成する．

この工程では全体の構造を決定したり，処理の流れなどを決め，実装するプログラミング言語に大きく依存しない設計を行う．そのため，この工程ではシステム全体のクラス構成や，タスクや資源のとり扱い方法など，実装するためにはじめに考慮しなくてはいけない範囲の設計モデルを作成する．また，分析工程にて決められた要求定義を実現するための，実装するプログラミング言語に依存しない範囲の処理の順序なども決める．

具体的に決められる設計制約の例としては，実現したい処理の手順や，タスクへのマッピングなどが含まれる．アーキテクチャ設計工程によって定義される設計制約の例は表 2.1 に挙げられるようなものが考えられる．

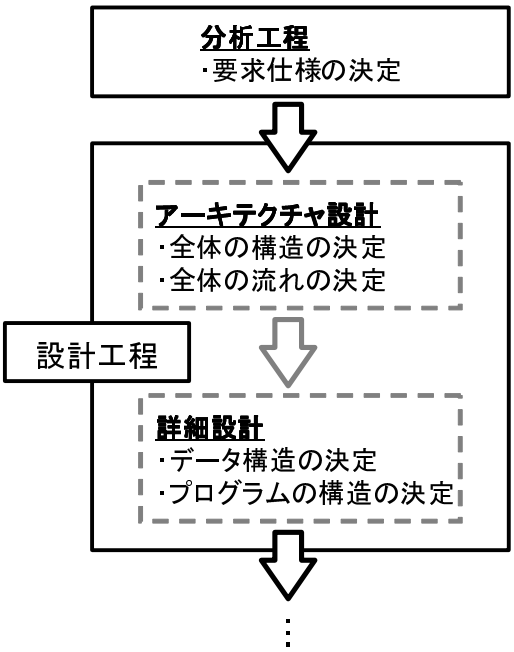


図 2.1: 開発工程の例

表 2.1: アーキテクチャ設計における設計制約の例

設計項目の例	設計制約の例	一般的な定義例
システム構成	システム全体のクラス構成	クラス構造や，関連への制約
タスクや資源管理	並行性 同期非・同期通信	用意するタスクの種類や，そのタスクの管理する機能 タスク間の通信方法や，共有資源の取り扱い方法
機能の実現方法	処理の順序	実現したい機能の具体的な処理の順序

2.2.2 詳細設計

詳細設計は，アーキテクチャ設計によって設計された設計モデルを，実装するプログラミング言語や実行環境を考慮した詳細設計モデルを作成する．

そのため，アーキテクチャ設計にて決められたクラスのインスタンスや，インターフェース，データ管理など，実行環境を考慮した詳細なシステム全体の設計モデルを作成する．また，アーキテクチャ設計にて決められた処理の順序を，実行環境や実装するプログラミング言語や実行環境を考慮したうえで，詳細に決める．

例として効率のよいメッセージ通信を行うためのデータ構造などが含まれる．設計制約の例は表 2.2 に挙げられたものが考えられる．

表 2.2: 詳細設計における設計制約の例

設計項目の例	設計制約の例	一般的な定義例
詳細なシステム構成	クラス間の関係	インスタンス インターフェース
詳細な処理の実現方法	プログラムの制御構造 データ構造	繰り返しや条件分岐 構造体やリストなど

2.3 設計制約とアスペクト指向の関係

前項にて整理した設計制約の例に多く共通している性質として，メッセージフローやデータフローが挙げられる．前項にて整理した設計制約のうち，これらメッセージフローやデータフローに関する設計制約について整理する．

整理した設計制約を表 2.3 に示す．メッセージフローやデータフローは複数のオブジェクトに横断的に存在するものであり，アスペクトとして捉えることに適している．そこで，メッセージフロー，データフローに関する設計制約を確認の対象とする．

表 2.3: メッセージフロー，データフローに関する設計制約

	設計工程	設計制約の例
メッセージフロー の存在する設計制約	アーキテクチャ設計	システム全体のクラス構成 並行性 処理の順序
	詳細設計	クラス間の関係 プログラムの制御構造
データフローの 存在する設計制約	アーキテクチャ設計	システム全体のクラス構成 処理の順序
	詳細設計	クラス間の関係

また，これらメッセージフローや，データフローなどは UML のシーケンス図で表現することが可能であるために，シーケンス図によって表現することのできる範囲を設計制約として検討を行う．シーケンス図は UML2.0 から大幅に拡張され，フレームや表現能力が豊かになっているため，UML2.0 に準拠した仕様とする．

また設計制約として，取り扱うメッセージフローやデータフローに関するもののうち，シーケンス図だけでは表現できないものがある．メッセージフローはオブジェクト間の相互作用として時間的前後関係がある．しかしその瞬間にのみ成立するような静的な関係の設計制約の場合，時間的前後関係がなくシーケンス図では表現できない．よってシーケンス図に OCL を組み合わせることで，静的な関係である設計制約を表現することとする．

第3章　アスペクト指向による設計制約の確認

3.1　概要

本研究にて確認を行う設計制約の性質には，メッセージフローとデータフローが含まれる．このうち，メッセージフローはシーケンス図により表現することができる．しかし，データフローに関しては，シーケンス図だけでは表現することはできない．シーケンス図だけでは表現できない理由としては，メッセージフローでは，各々のメッセージには時間的な前後関係が設計制約の中に存在するためにシーケンス図で表現できるが，データフローには時間的前後関係が設計制約に存在しないことが挙げられる．

しかし，データフローにて，データの変化が，メッセージの呼び出しを基準にして変化を起こす事に着目すると，シーケンス図中のメッセージの前後にて変化する値を確認することで，データフローの確認をすることが出来る．

そのために，メッセージフローで表現される時間的な設計制約をシーケンス図で表現し，そのメッセージの呼び出される瞬間的な設計制約はOCLで表現することとする．

シーケンス図とOCLで表現される，設計制約の関係を図3.1に示す．

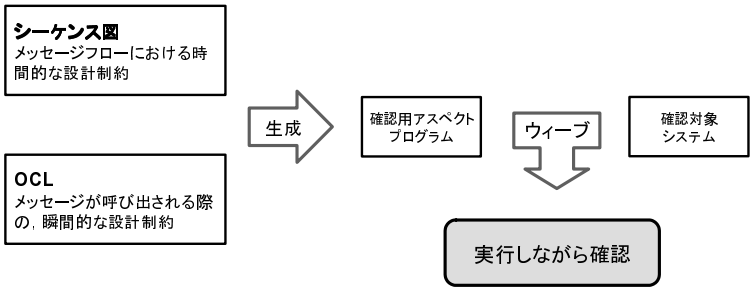


図 3.1: シーケンス図とOCLで表現される設計制約と確認方法の関係

3.2　アスペクトを用いることの利点

アスペクト指向言語を使用する利点は，はじめに組み込みシステム特有のクロス環境とセルフ環境の問題に対応できることである．クロス環境ではアスペクトを使用し，セル

フ環境ではアスペクトを外すといった，組み込みシステムの開発環境に対応できることが挙げられる．通常，クロス環境上で確認する場合，クロス環境用の確認プログラムを記述し，セルフ環境上で実行する場合には，クロス環境用のプログラムを外すなどの必要性が生じる．しかし，アスペクトを使い分けることで，クロス環境用のプログラム，セルフ環境用のプログラムの両方を容易に作成することが出来る．

次にプログラム中に直接記述しないで，機能を織り込むことができる点が挙げられる．例えば，ある機能が実行されたことを確認する場合を考える．ある機能が実行された時にログファイルとして出力する機能をアスペクトとして扱うことで，実際のプログラムに直接記述しないで機能を追加できる．このようにアスペクトを用いることで，可読性を下げないという利点がある．

最後にアスペクトを切り替えることで，異なる制約確認機能を織り込むことができることが挙げられる．これは確認したい性質が複数ある場合，それら各々を別のアスペクトとして記述することで容易に確認項目を切り替えることができる．またこの場合も同様に，実際のプログラムに記述することなく行える．

3.2.1 設計制約確認方法の位置づけ

本研究における確認方法は，高機能なアサーションのような機能として位置づける．従来のアサーションとの相違点は以下の点である．

- プログラムに直接記述することなく機能を織り込むことができるために，可読性に影響を与えない．
- 単純にプログラム上のその位置に到達したら，アサーションが作用するのではなく，作用するための前提条件などを指定できる．
- アサーションよりもより複雑な機能を織り込むことができる．
- アサーションではどこでも記述できたが，アスペクト指向言語を用いる際には Joinpoint 以外には作用させることは出来ない．

3.3 シーケンス図にて表現される設計制約

3.3.1 概要

本項では，シーケンス図にて表現される設計制約確認方法に関して述べる．シーケンス図にて表現される設計制約は処理の順序の確認であり，その方法について述べる．

処理の順序を確認するためには，以下の手順にて行う．

1. シーケンス図で記述される設計制約を入力とする．

2. 入力されたシーケンス図から，図中に記述されている処理の順序を確認するためのアスペクトを出力する．

3. 出力されたアスペクトと，確認を行うプログラムをウィーブし，実際に実行させながら確認を行う．

ここで入力されるシーケンス図には，対象とする機能の中で，確認したいメッセージの相互作用を記述する．処理の順序の確認を行うアスペクトには，シーケンス図に記述されるメッセージが，実際に呼び出される際に，メッセージ固有の ID をスタックに積むための仕組みが実装される．積まれたスタックは更新される度に，あらかじめ用意される処理の順序を確認すための正規表現とマッチするか比較される．あらかじめ用意される正規表現には，メッセージが呼び出される際に積まれる ID 列とマッチする正規表現を用意する．正規表現の確認には，正規表現確認用のライブラリを用意し，確認を行うこととする．

正規表現を用いて，処理の順番を確認することの利点は，第 1 に確認できる範囲が正規表現にて表現が可能な範囲となることが挙げられる．第 2 にシーケンス図にてフレームを扱う場合に，メッセージの順番に入れ子構造が生じるが，正規表現では入れ子構造の確認が出来ることである．第 3 に，同じメッセージが複数回呼ばれるときに，正確に確認を行うことが出来るという点である．

シーケンス図のフレームによる入れ子構造とは，シーケンス図では様々な性質をフレームとして表現することが可能であるが，このフレームが入れ子構造をとることである．フレームによる入れ子構造がある場合，階層が下のフレームから処理されるために，入れ子構造を表現できる正規表現を選択した．

また，メッセージが複数回呼び出される際に正確に確認が行うことが出来るとは，以下のような要因からである．アスペクトは基本的には，1 つの JoinPoint には 1 つのアスペクトしか記述出来ないので，何回呼び出されても同じアスペクトが呼び出されるだけである．そのため，何度も同じメッセージが呼び出されても，実行されるアスペクトは 1 つであり，呼び出しの度に処理を変えることができない．しかし，正規表現による確認方法では，同じメッセージが複数回呼び出されても表現することが可能である．例えば msg1，msg2，msg1 と呼び出される場合，msg1 に a，msg2 に b という ID を振り，正規表現で”aba”と表現ができ，確認することができる．

他に処理の順序を確認する方法として，フラグによる確認方法も検討を行った．フラグによる確認方法とは，メッセージが呼び出される際にフラグを更新していく方法である．しかしフラグによる確認方法は，入れ子構造に対応していなく，また同じメッセージが複数回呼び出されても正確に確認することができない．そのため，フラグ方式ではなく正規表現による確認方法を選択した．

3.3.2 表現方法と確認方法の制限

アスペクト言語は設計制約として記述するメッセージすべてにフックするため，同じプログラム中に同じメッセージを呼び出す箇所全てが JoinPoint として成立する．同じ名前

のメッセージであれば全てフックするので，着目したメッセージは，確認したい機能中に含まれる回数，過不足なく記述する必要がある．

これは，確認対象の機能中に実際には 2 回呼ばれているにも関わらず，設計制約としてのシーケンス図に 1 回分しか記述していない場合には確認できなくなるからである．

3.3.3 各性質の表現方法と確認方法

本項では処理の順序の確認において，シーケンス図にて表現される性質の表現方法と，確認方法に関して述べる．シーケンス図にて表現される性質は，様々なフレームにて性質を表現する．よってフレーム毎に，その性質，表現方法，確認方法をまとめる．これよりフレームの説明では，以下の書式により説明を行う．

- フレーム名
  フレームの名称
- 説明
  フレームの表現する性質について説明を行う．
- シーケンス図による表現の例
  図は例を用いて説明する．
- メッセージフロー
  フレームで表現されるメッセージフローについて説明する．またシーケンス図に記述された例を用いて説明する．
- スタックの例
  正規表現と確認のために，積まれるスタックを説明する．またシーケンス図に記述された例を用いて説明する．
- 正規表現の例
  確認を行うための正規表現を説明するまたシーケンス図に記述された例を用いて説明する．

フレームの説明において，以後フレームをさらに分けた部分を上から順に，”**領域 1**”，”**領域 2**”と”**領域＋領域数**”で呼ぶ．領域は，分割された数だけ存在する．また，フレーム全体を表すときには [**フレーム名**] で表現する．

フレーム名：strict

フレームの説明：

一番上位のフレームを sd(SequenceDiagram) とする．sd フレーム内のメッセージは strict とする．strict とは領域内の相互作用に強い順序性が存在することを表しており，強い順序性とは，シーケンス図に記述されるメッセージが上から順々に呼び出され，その順番が変化することが無いことを表す．以下，領域内のメッセージに強い順序性のある場合には，[strict] と表現する．

シーケンス図の例

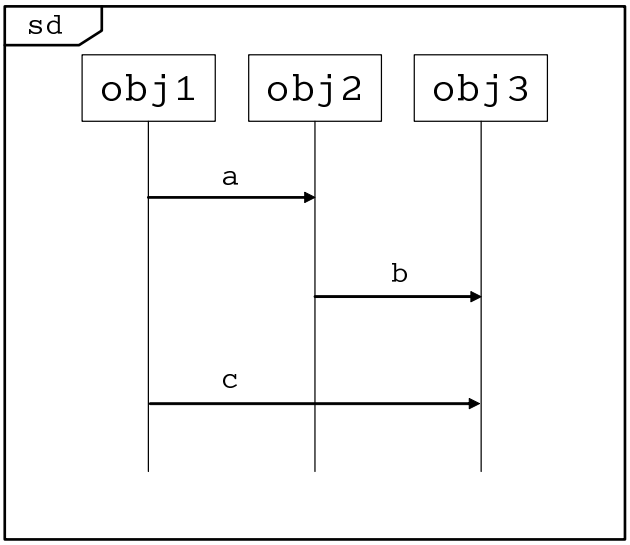


図 3.2: strict の例

上から順に，呼び出されるメッセージを記述する．記述されるメッセージは，同じタイミングで呼び出されることは無いために，メッセージの呼び出される順番は一意に決定する．

メッセージフロー

上から順に行われる．

- 例：a ⇒ b ⇒ c

スタック

呼び出される順に積まれる.

- 例：abc

正規表現

シーケンス図の上から順に並べたものであり，sd フレームは [strict] である.

- 例：abc

フレーム名：alt

フレームの説明：

alt フレームは条件分岐を表現する．フレーム内において点線で区切られた領域のうちどれか一つが実行される．UML2.0 の仕様ではさらに分岐条件を記述できるが，本研究では”分岐している”ことを確認するために，分岐条件は無視することとする．またフレーム内の各領域のメッセージは strict である．

シーケンス図の例

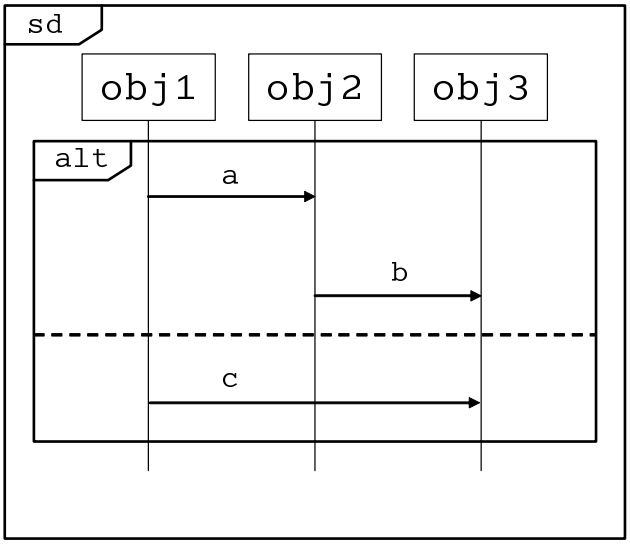


図 3.3: alt の例

点線で区切られた領域のどれかが実行されることを表す．また，フレームと同じタイミングで呼び出されるメッセージは無いものとする．

メッセージフロー

領域 1 or 領域 2 or ･･･

点線で区切られた領域が 2 つ以上ある場合は，その 3 つ全てを ”or” の関係で表現する．

- 例：a ⇒ b or c

スタック

領域 1 or 領域 2 or ･･･

同様に，点線で区切られた領域が 2 つ以上ある場合は，その 3 つ全てを ”or” の関係で表現する．

- 例：ab　or　c

正規表現

(領域 1| 領域 2| ･･･)

同様に，点線で区切られた領域が 2 つ以上ある場合は，その 3 つ全てを ”or” の関係で表現する．また領域内は [strict] である．

- 例：ab|c

フレーム名：loop

フレームの説明：

loop はフレームで囲まれた領域が繰り返すことを表現する．UML2.0 の仕様ではさらに繰り返し条件を記述できるが，本研究では”繰り返す”ことを確認するために，繰り返し条件は無視することにする．またフレーム内のメッセージは strict である．

シーケンス図の例

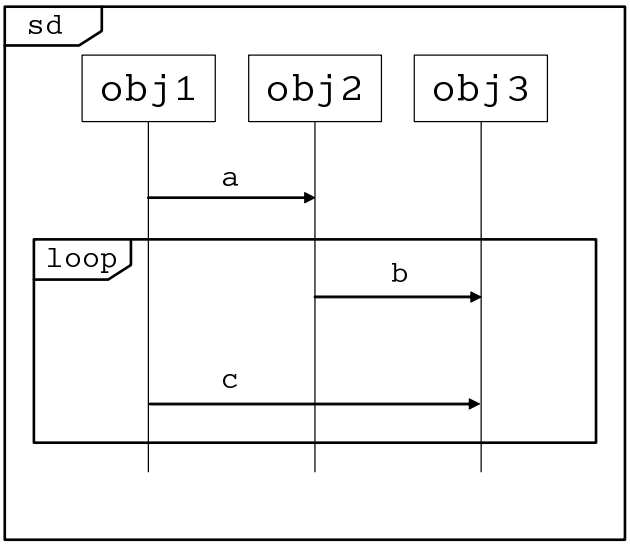


図 3.4: loop の例

loop フレーム内のメッセージが繰り返すことを表現する．他のフレーム同様に，フレームと同時呼び出されるメッセージは無いものとする．

メッセージフロー

loop フレーム内メッセージが，繰り返される．

- 例：a ⇒ b ⇒ c ⇒ b ⇒ c ⇒ b ⇒ c ⇒ ··· ⇒ b ⇒ c

スタック

loop フレーム内メッセージが，繰り返される．

- 例：abcbcbcbcbc

正規表現

[loop] ＊

領域内のメッセージの順番は [strict] である．正規表現にて確認する性質は，繰り替えしが行われた，ということであり，回数や上限などは指定しない．

- 例：a(bc)∗

フレーム名：par

フレームの説明：

par フレームは並行動作を表現するため，点線で区切られた領域が各々並行に動作することを表現している．そのため par フレームの確認には，スタックを領域の数用意し，各々のメッセージを積み，各々のスタックに対応した正規表現で確認を行う．全ての領域の確認が完了することを持って，par フレームの確認をとることが出来る．また par フレームの各領域のメッセージも strict である．

シーケンス図の例

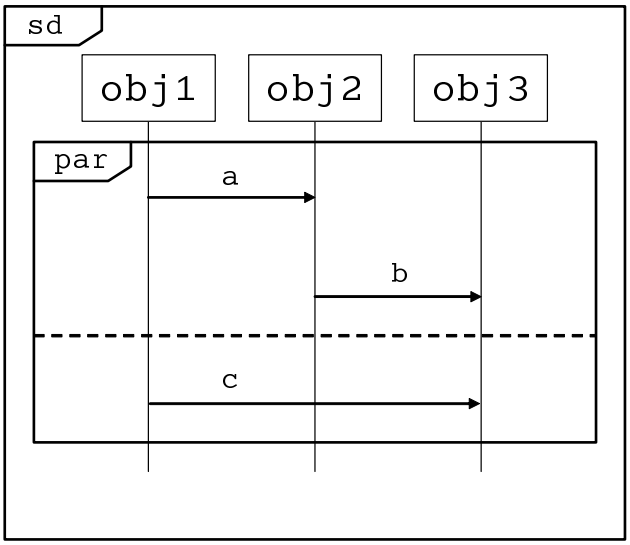


図 3.5: par の例

par フレーム内の各領域が，各々並行動作することを表現する．他のフレーム同様に，フレームと同時呼び出されるメッセージは無いものとする．

メッセージフロー

各々の領域内のメッセージの順番は，上から順に行われるが，領域の異なるメッセージの順番は，制限されない．そのため各領域にて決められた順番における組み合わせた結果が実行される．

- 例：a ⇒ b ⇒ c　or　a ⇒ c ⇒ b　or　c ⇒ a ⇒ b

スタック

(領域 1 で決められた順番) , (領域 2 で決められた順番) ⋯
区切られる領域が 2 つ以上の時は，領域 3 ⋯ と続く.

- 例：ab (上の領域) , c (下の領域)

正規表現

par フレームは領域を複数持つために，各々スタックを持つ．各々のスタックを正規表現で確認を行い，マッチすることでフラグをたて，その and 条件で par フレームとしての確認を行う．各々の領域の正規表現は [strict] で与えらる.

- 例：ab (上の領域) c (下の領域)

フレーム名：seq

フレームの説明：

seq フレームは領域内のメッセージの弱い順序性を表現する．seq フレームには以下のように定義される．

- 点線で区切られたの領域のメッセージの順番は保存される．
- 異なる領域の，異なるライフライン上のメッセージの順番は保存されない．
- 同じライフライン上のメッセージの順番は保存される．

これら seq フレームの確認には，はじめにスタックをライフラインの数用意する．次に各々のメッセージの両端を ID とし，その ID をスタック積み，各々のスタックに対応した正規表現で確認を行う．各ライフライン上の確認が完了することを持って，seq フレームとしての確認をとることが出来る．

シーケンス図の例

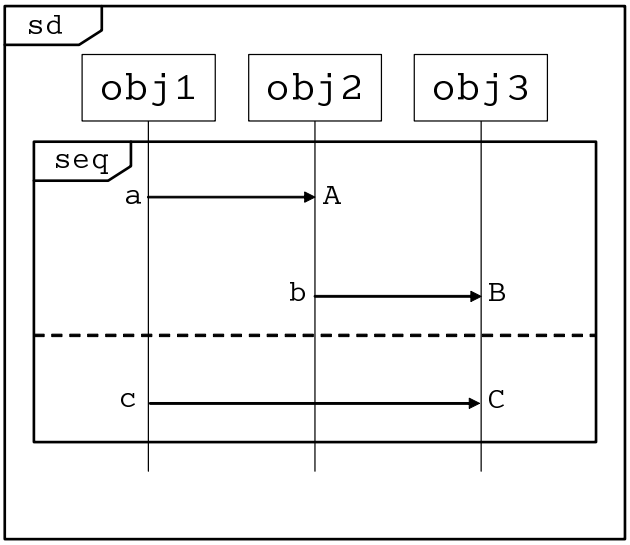


図 3.6: seq の例

他のフレーム同様に，フレームと同時呼び出されるメッセージは無いものとする．

メッセージフローの例

seq フレームにおけるメッセージの送受信は非常に複雑になるために，図 3.6 の例で，起こりうるすべての場合を図 3.7 に示す.

- a ⇒ A ⇒ b ⇒ B ⇒ c ⇒ C
- a ⇒ A ⇒ c ⇒ b ⇒ B ⇒ C
- a ⇒ c ⇒ A ⇒ b ⇒ B ⇒ C

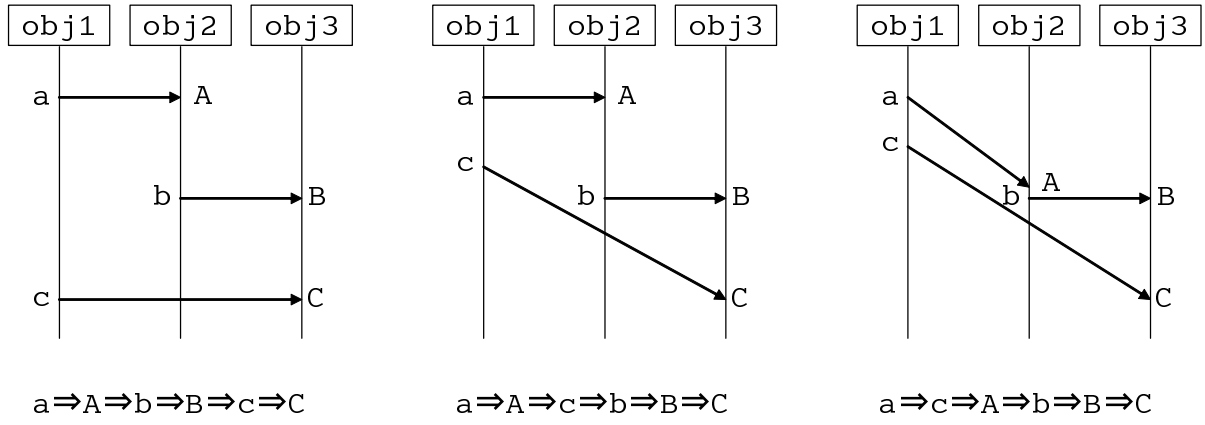


図 3.7: seq の例 2

スタック

各オブジェクトに積まれるスタック，各オブジェクトライフライン上に存在するメッセージの端点が，上から順に積まれる.

- obj1 のスタック例：ac
- obj2 のスタック例：Ab
- obj3 のスタック例：BC

正規表現

seq フレームも par フレームと同様に，スタックを複数持つ．各々のスタックを正規表現で確認を行い，マッチすることでフラグをたて，その and 条件で seq フレームとしての確認を行う．各々のスタックに対する正規表現は，各オブジェクトのライフライン上にあるメッセージを上から順で与えられる．ライフライン上におけるメッセージの順番は [strict] となる.

- obj1 のスタック例：ac
- obj2 のスタック例：Ab
- obj3 のスタック例：BC

入れ子構造

説明：

シーケンス図では入れ子構造をとることができる．本研究では”alt フレーム”，”loop フレーム”に関しては，確認する際に入れ子構造をとらない定義をした．実際に入れ子構造をとるのは”par フレーム”と”seq フレーム”の 2 つになる．ただし表現としては全てのフレームが入れ子構造表現を行う．入れ子構造をとる場合，フレームは一つ上位側から見た場合，その子フレームは一つのメッセージとして扱う．子フレーム単体の確認が取れ次第，上位側のメッセージ付加した ID とは異なる一つの ID を振りわける．新たに振った ID を上位側の正規表現で確認を行う．

シーケンス図の例

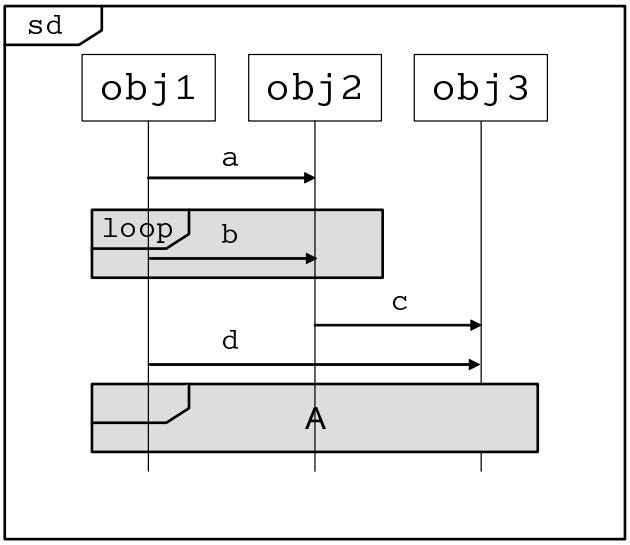


図 3.8: 入れ子の例

メッセージフロー

sd フレームにおけるメッセージフローは，[strict] であり，入れ子状態としてある par フレームや seq フレームは，一つのメッセージとして処理される．

- 例：a ⇒ b ⇒ b ⇒ b ⇒ b ⇒ ⋯ ⇒ c ⇒ d ⇒ A

スタック

par フレーム，もしくは seq フレームがフレーム単体として確認を行い，フレームに対し振り分けられる固有の ID が，上位側のフレームに積まれる．上位側のフレームにおけるメッセージの順番は [strict] となる．

- 例：abbbbbcdA

正規表現

par フレーム，もしくは seq フレームを一つのメッセージと見なし，固有の ID を振り分ける．その上で，上位側のフレームを [strict] として正規表現で表す．

- 例：ab∗cdA

3.4 OCLにて表現される設計制約

3.4.1 概要

シーケンス図はメッセージの送受信の相対的な前後関係を示すものであるが，そのメッセージの送受信の瞬間的なデータ間の制約を表現することはできない．そこで，シーケンス図では表現できないような静的な関係の設計制約について，OCL を用いることで表現することとする．

OCL を用いて確認をする静的な関係の設計制約とは，例えばメッセージの送受信に伴い，その前後で変化する値などの関係である．OCL を用いることで，様々なタイミングにおける変数の値を表現することが出来るために，データフローなどの確認をすることが出来る．変数の値は，OCL を用いてメッセージの送受信の前後にどのような条件が存在するかを制約として与えることで確認することとする．

またシーケンス図に OCL を用いて設計約を表現することで，様々な設計制約を確認することができる．本研究では排他性の制御に用いられるセマフォに注目して，シーケンス図と OCL を用いた応用とした位置づけで確認を行うこととする．

3.4.2 確認対象

OCL で表現し，確認するのはメッセージ送受信の際の返り値，引数であり，条件は以下の 3 つとする．

- inv：不変条件
- pre：事前条件
- post：事後条件

3.4.3 表現方法と確認方法の制限

変数をアスペクト言語を用いて確認する場合には，プログラムすべての状態を取得する事は出来ない．取得できるのは JoinPoint としてフックできる箇所に限定される．今回使用する AspectC++では変数を JoinPoint とすることが出来ないため，確認対象とする変数は AspectC++で取得することのできる引数と返り値に限定する．一方 AspectJ などでは変数の変化に対しても JoinPoint とすることが出来るために，確認できる範囲は広く行うことが可能と思われる．

また通常，不変条件とした場合，その確認対象としている間の状態全てにおいて満たすべき条件である．しかし本研究ではメッセージの送信時，終了時の瞬間において満たすことが条件となる．よってあるメッセージの変数に関して不変条件を設定した場合には，呼び出されたメッセージの送信時と終了時に満たしているかを確認することとなる．

3.4.4 表現方法

変数の確認にはOCLを用いる．OCLでは以下のような書式で記述する．

context :: [object name] : [message name]
[parameter] : [fomula]

* parameter = inv, pre, post

確認対象とする3条件は，1つのcontextの中に記述できる．また実際の使用例を図3.9とOCLで示す．

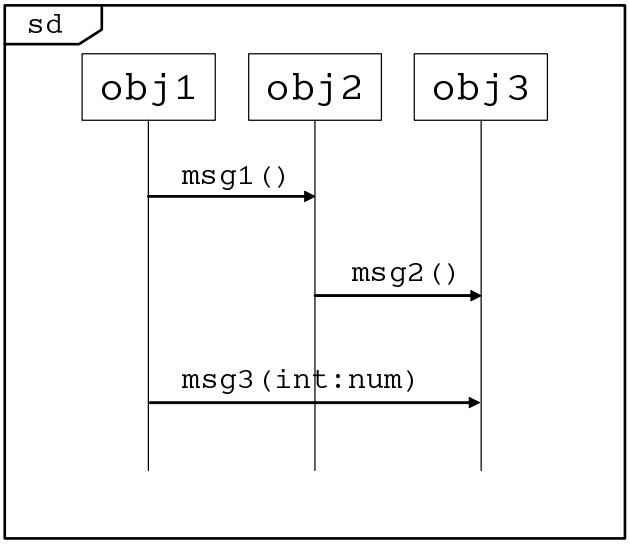


図 3.9: 変数の確認の例

context :: obj2 : msg3(int:num)
pre : num > 0

上記のシーケンス図とOCLで表現している設計制約は，obj2オブジェクトの，msg3メッセージの引数numは，事前条件として[num > 0]という条件である．

3.4.5 確認方法

OCL にて記述された変数への設計制約は以下の表 3.1 のようにアスペクト言語へマッピングされ，確認を行う．この方法にて変数を確認できるタイミングは，メッセージの送信

表 3.1: OCL とアスペクト言語との対応

OCL	アスペクト言語
inv(不変条件)	JoinPoint の before と after の組み合わせにて確認
pre(事前条件)	JoinPoint の before にて確認
post(事後条件)	JoinPoint の after にて確認

時や終了時である．そのライフライン上で全てで OCL にて記述した制約が有効では無く，そのメッセージの呼び出し時のみである．そのため”pre(事前条件)”と”post(事後条件)”両方で確認を取るとした”inv(不変条件)”では，メッセージの送信時から終了時の間に値が変化しても，送信時と終了時にさえ値を満たしてさえいれば”不変である”とする．

3.5 シーケンス図と OCL により表現される設計制約の応用例

3.5.1 セマフォの確認

シーケンス図と OCL を用いて表現を行う設計制約の応用例として，セマフォに注目する．セマフォの確認は，設計制約を記述するシーケンス図だけでは記述することが出来ないために，シーケンス図とセマフォの制約を記述した OCL を組み合わせることで確認を行う．シーケンス図にて，セマフォのためのメッセージの処理の順番の制約を表現し，OCL にてそのメッセージの送受信の際に与えられる制約を表現する．このシーケンス図と OCL の区切られる領域が 2 つ以上の時は，領域 3 … と続く組み合わせでセマフォの確認を行う．確認を行うために制約を与える対象は，共有資源を持つオブジェクトであるため，共有資源にアクセスする全てのオブジェクトに制約を与える必要がない．その結果セマフォの確認を行うための設計制約を容易に設定することが出来る．

もし共有資源にアクセスするオブジェクト全てに制約を与えてしまうと，確認を行うための制約の記述事態に誤りが入り込む恐れがある．この問題を解決するため，共有資源自身のみに制約を与えることで確認をとる方法を選択した．

3.5.2 確認対象

セマフォの確認とは，セマフォとして共有資源にアクセスする際の約束事が守られているかを確認する．約束事とは，共有資源にアクセスする前に，セマフォを取得したものだ

けがアクセス権を得ること．共有資源へのアクセスが終了し次第，セマフォを解放する，という 2 つの事柄である．

そこでセマフォの確認とは，以下の 2 点をもって，セマフォの確認の確認をとることとする．

- 共有資源にアクセスするオブジェクトは，最も最近セマフォを取得している
- セマフォを解放するオブジェクトは，最も最近共有資源にアクセスしていた

3.5.3 表現方法と確認方法の制限

セマフォを確認するためにはセマフォの取得と解放，共有資源へのアクセスを JoinPoint としなくてはいけない．しかしアスペクト言語はメッセージを JoinPoint とするので，セマフォの取得，解放，共有資源へのアクセスは全て関数として存在する必要がある．また，セマフォを確認するための制約を記述するために OCL を用いる．しかし，標準的な OCL では記述できないので，OCL の拡張を行う．OCL の拡張については次項にて説明する．

3.5.4 表現方法

セマフォを確認するための設計制約は，シーケンス図と OCL にて表現する．はじめにセマフォの表現に用いるシーケンス図は図 3.10 となる．

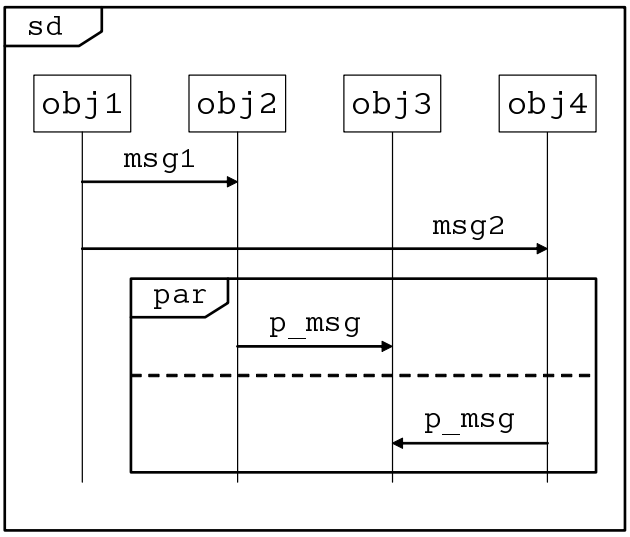


図 3.10: セマフォの例

このシーケンス図だけでは，p_msg は並行の動作する，という状態を表現しているだけである．そこで OCL を組み合わせてセマフォを確認するための設計制約を表現する．組み合わせる OCL は図 3.10 の場合は以下のようになる．

context :: obj3 : p_msg()
- - 共有資源にアクセスしたオブジェクトは,
- - 最も最近セマフォを取得しているオブジェクトと同一である
pre : obj2.sem_get().getCallObj() = obj2.p_msg().getCallObj()
- - セマフォを解放したオブジェクトは,
- - 最も最近共有資源にアクセスしたオブジェクトと同一である
post : obj2.p_msg().getCallObj() = obj2.get_sem().getCallObj()

この OCL では，共有資源を持っている obj3 のセマフォを取得，解放したオブジェクト，共有資源にアクセスしたオブジェクトを取得できるように，”getCallObj()”という関数を OCL に拡張した．これにより，共有資源へのアクセスを行ったオブジェクトの情報を取得することが可能になった．

3.5.5 確認方法

セマフォのシーケンス図と OCL によって表現し，確認するための設計制約の確認方法は以下の手順により行われる．

1. セマフォを取得されたら，セマフォ取得したオブジェクトを取得する
2. 共有資源へアクセスされたら，アクセスしたオブジェクトを取得する
3. 共有資源へアクセスしたオブジェクトと，セマフォを取得したオブジェクトを比較する．
4. セマフォを解放されたら，セマフォを解放したオブジェクトを取得する
5. セマフォを取得したオブジェクト，セマフォを解放したオブジェクト，共有資源にアクセスしたオブジェクトを比較し，全て同一のオブジェクトであることを確認する

以上の方法により，セマフォの確認をとることが出来る．

第4章　設計制約確認ツール

4.1　概要

設計制約確認ツールは，前章までに述べた確認方法を用いて，設計制約を容易に確認することができるかを評価するために作成した．設計制約確認ツールへの入力は，設計制約であり，出力は設計制約を確認するためのアスペクトプログラムとなる．設計制約とツールの関係を図 4.1 に示す．設計制約はシーケンス図と OCL で記述され，設計制約確認ツールに読み込まれる．設計制約確認ツールにて読み込まれた設計制約から，設計制約を確認するためのアスペクトプログラムを出力する．出力されたアスペクトプログラムは，用意されている正規表現ライブラリを用いて確認動作を行う．

出力された AspectC++プログラムを，確認対象のプログラムにウィーブすることで，設計制約を確認するプログラムが埋め込まれたプログラムが作成される．このウィーブされたプログラムを実行することで，確認対象の機能を確認することができる．

また本ツールは，確認を行う対象のシステムの全ての処理の順番を確認するものではなく，確認を行いたい部分のみに着目するものである．そのため，記述するメッセージも，確認を行う機能の主要な部分を記述し，機能の実現に影響の少ないメッセージは記述しない．ツールとしては，記述されているもののみを確認する．

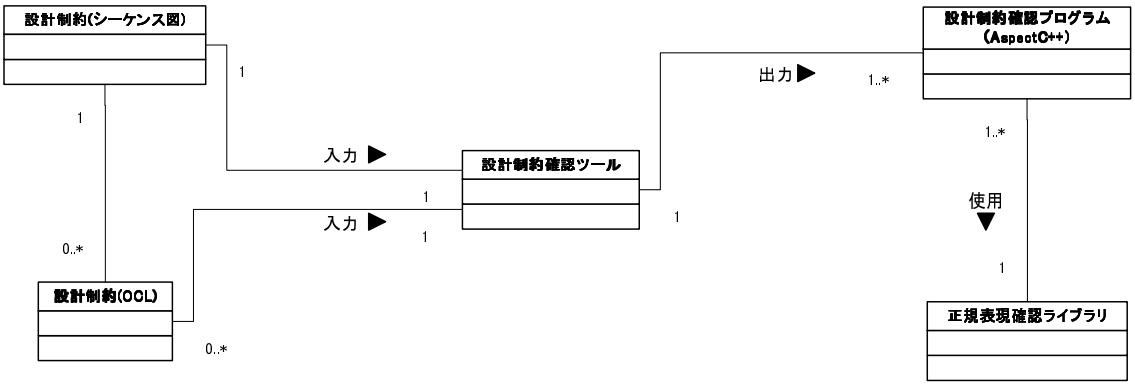


図 4.1: 設計制約とシステム構成

4.2 設計制約確認ツールの構成

本ツールの入力は，確認対象とするシステムに対して，確認したい性質を記述した設計制約であり，これはシーケンス図と OCL で構成される．入力されるシーケンス図は XML 形式で記述されており，OCL はツール上のエディタで直接記述される．

入力された設計制約を解析し，設計制約を確認することのできる AspectC++プログラムを出力する．出力された AspectC++プログラムは，確認対象のシステムにウィーブされ，実際に動作させることで確認を行う．確認を行う際には同時に用意された正規表現確認用ライブラリを用いて行われる．正規表現ライブラリは POSIX で定義された regex を用いて作成した．

設計制約とシステム構成の関係を図 4.2 に示す．

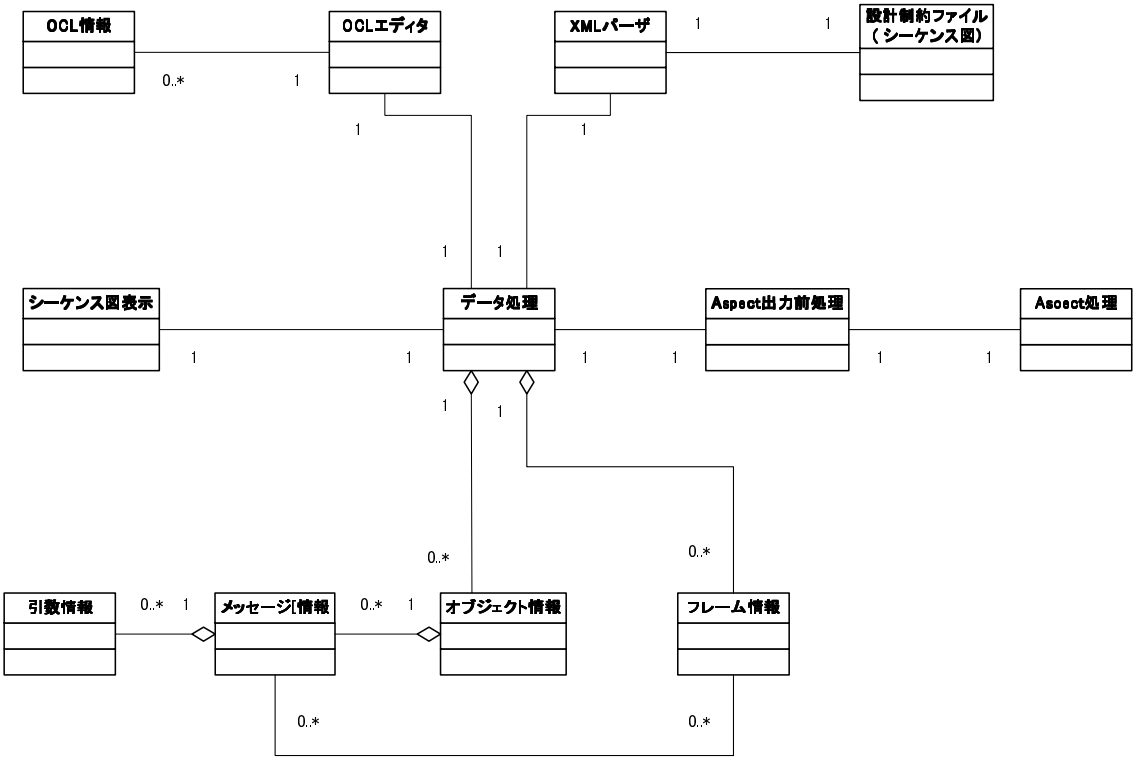


図 4.2: 設計制約確認ツールのシステム構成

4.3 設計制約データ

設計制約はシーケンス図と OCL にて構成される．本項ではこれらの構成について説明する．

- シーケンス図 –XML–

シーケンス図は以下の XML により構成される．各オブジェクトに所属するメッセージ，そのメッセージに保持される情報などが記述される．また，フレーム情報もこの XML に含まれる．

```
<sd>
  +– <title> 属性名：name シーケンス図の名前
    +– <obj> 属性名：name, id オブジェクトの名前，固有の ID No.
      +– <active> 属性名：id メッセージのあるアクティブな区間の ID No.
        +– <msg> 属性名：name, id メッセージの名前，固有の ID No.
        +– <result> 属性名：name, form 返り値の名前，返り値の型
        +– <args> 属性名：name, form 引数の名前，引数の型
        +– <target> 属性名：targetname, objid, targetid, return
          送信先のオブジェクトの名前，送信元オブジェクトの ID,
          送信先の Active ID，return があるかどうか
    ……
  +– <frame> 属性名：name フレームの名前
    +– <index> 属性名：obj, objid, name, msgid, targetobj
      送信先オブジェクト名，送信先オブジェクト ID，送信メッセージ名,
          送信メッセージ ID，送信先オブジェクト ID
```

- OCL
  変数やセマフォの確認を行うために，設計制約を OCL にて記述する．記述には本ツール上のエディタにて直接記述する．変数の確認のための設計制約記述には，確認対象のオブジェクト，メッセージ，変数の名前，確認条件を記述する．セマフォの確認のための設計制約記述には，確認対象のオブジェクト，メッセージ，を記述する．

4.4 設計制約確認ツール

本項では設計制約確認ツールについて述べる．本ツールの機能は，設計制約を解析し，確認をするための AspectC++プログラムを出力するものであり，その機能について説明する．

4.4.1 構成

本ツールは図 4.2 で示した構成になる．設計制約確認ツールは Java で作成したため，XML パーザには SAX(Simple API for XML) を用いた．XML で記述された設計制約ファ

イルよりシーケンス図を描画し，処理の順番を確認する AspectC++プログラムを出力する．また，ツール上の OCL エディタに記述された変数の確認を行うための設計制約からも，AspectC++プログラムを出力する．同時にエディタにてセマフォを確認するための OCL 記述がある場合にはセマフォを確認するための AspectC++プログラムを出力する．処理の順序，変数の確認，セマフォの確認を行う場合，各々独立した AspectC++ファイルを出力する．

4.4.2 データ構造

解析される設計制約は，ツール内部にて，フレーム情報–オブジェクト情報–メッセージ情報–引数情報，からなる構成になっているために，以下の図 4.3 ようなデータ構造をもつ．各フレームに属したメッセージなどの相互関係から，対応した AspectC++プログラムを生成する．

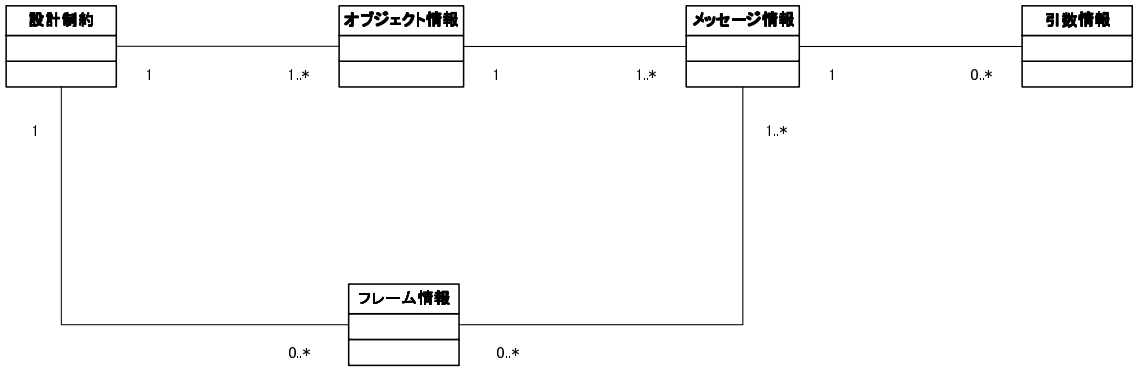


図 4.3: データ構造

4.5 正規表現ライブラリ

処理の順番を確認するために使用される正規表現ライブラリは，POSIX にて定義される regex を使用して作成した．ライブラリにて定義される正規表現を確認するための関数は以下のようになる．

- **bool** *pattarnMatch*(**char*** pattarn, **char*** str, **char*** state);

pattarn : 正規表現
正規表現で与えられる文字列は，確認対象の処理の順序とマッチすべき正規表現である．

str : 確認対象文字列
スタックに積まれた文字列であり，pattarn によって与えられる正規表現とマッチするか確認を行う対象である．

state：確認フレーム名
現在確認中のフレームである．これは現在確認中のフレームが par，seq，strict，のいずれかを表示するための値である．

返り値の型は **boolean** であり, 与えられた正規表現と確認する文字列がマッチした場合に **true** を返し，マッチしなかった場合には **false** を返す．

4.6 出力されるアスペクトプログラムの例

本ツールによって出力される，設計制約を確認するための AspectC++プログラムの例を示す．はじめに例とする設計制約は図 4.4 であり，変数の確認，セマフォの確認は以下のように与えられる．

- 変数の確認例

> **context :: obj4 : msg2(int : num)**
> - - 変数の確認，*num* は *0* 以上である
> **pre : num > 0**

- セマフォの確認例

set() に関する OCL

> **context :: obj3 : set()**
> - - 共有資源にアクセスするオブジェクトは,
> セマフォを取得しているオブジェクトと同じである
> **pre : obj3.get_sem().getCallObj() = obj3.set().getCallObj()**
> - - 共有資源にアクセスしていたオブジェクトは,
> セマフォを解放したオブジェクトと同じである
> **post : obj3.set().getCallObj() = obj3.free_sem().getCallObj()**

get() に関する OCL

> **context :: obj3 : get()**
> - - 共有資源にアクセスするオブジェクトは,
> セマフォを取得しているオブジェクトと同じである
> **pre : obj3.get_sem().getCallObj() = obj3.get().getCallObj()**

- - 共有資源にアクセスしていたオブジェクトは,
　セマフォを解放したオブジェクトと同じである
post : obj3.get().getCallObj() = obj3.free_sem().getCallObj()

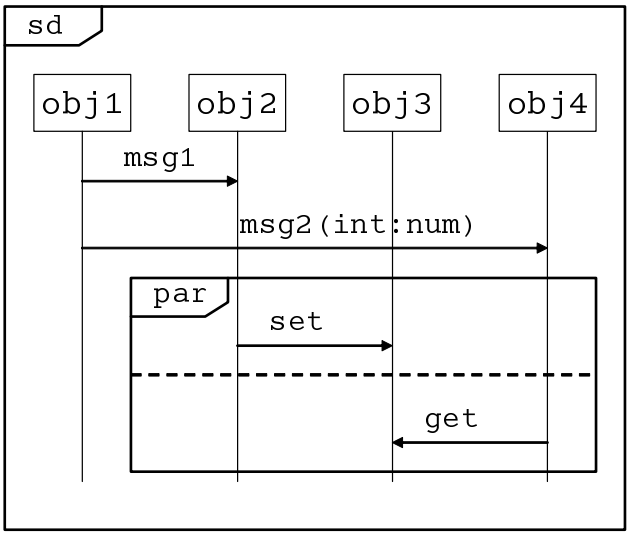


図 4.4: 設計制約の例

例題では，以下の 3 つの性質を確認する．

1. par フレームを含む処理の順番
2. msg2 における引数の値の取りうる範囲の確認
3. obj2 と obj4 からの set()，get() がセマフォを守ってアクセスをしているか

また，処理の順序の確認する AspectC++プログラムを以下に示す．ただし，確認のための出力など，確認動作に直接関係の無いところは省略した．

- 処理の順序

処理の順序を確認するための AspectC++プログラムでは，はじめに確認を行うためにシーケンス図に記述されたメッセージが呼び出されると，順番にメッセージ固有の ID をスタックに積む．次に正規表現確認ライブラリにて確認を行う．

par フレームでは，はじめに点線で区切られた領域各々に用意されたスタックに順番に積む．次に正規表現ライブラリにて確認し，確認をとることができればフラグを立てる．他の領域も同様に確認を行い，各領域で立てられたフラグが全て変更されたことを持って，par フレームの確認を取ることができる．

par フレームの確認を取り，1 つ上位のフレームのスタックに，フレームに振り分けられた ID を積む．例では”0”という ID を積んでいる．

```
aspect Trace{

private:
    char stack[100], pattern[100];
    char par01[100], par02[100], parpat01[100], parpat02[100];
    int parflag01, parflag02

public:
    Trace(){
        strcpy(pattern, ”ab0”);
        strcpy(parpat01 , ”c”);
        strcpy(parpat02 , ”d”);
    }

    advice call(”% msg1()”) : before(){
        strncat(stack, ”a”, 1);
        pattarnMatch(stack, pattern, ”strict”);
    }

    advice call(”% msg2(%)”) : before(){
        strncat(stack, ”b”, 1);
        pattarnMatch(stack, pattern, ”strict”);
    }

    advice call(”% set(%)”) : before(){
        strncat(par01, ”c”, 1);
        if(pattarnMatch(par01,parpat01,”par01”){
            parflag01 = 10;
            if(parflag01 == parflag02){
                strncat(stack, ”0”,1);
                pattarnMatch(stack, pattern, ”strict”);
            }
        }
    }

    advice call(”% get(%)”) : before(){
        strncat(par02, ”d”, 1);
        if(pattarnMatch(par02,parpat02,”par02”)){
            parflag02 = 10;
            if(parflag01 == parflag02){
                strncat(stack, ”0”,1);
                pattarnMatch(stack, pattern, ”strict”);
            }
        }
```

```
    }
};
```

- 変数の確認

変数の確認するための AspectC++プログラムでは，JoinPoint API にて引数を取得する．取得した引数を，OCL にて記述された条件で比較することで確認を取る．例では取得した引数は”num>0”という条件を，advice の中で確認を行っている．確認を取り，標準出力に出力する．

```
aspect OCLcheck{
    advice call("% msg2(int)") : before(){
        int *argname;
        argname = (int*)tjp->arg(0);
        if(*argname>0){
            printf("\t>> OCL CHECK OK! \n");
        }
    }
};
```

- セマフォの確認

今回は並行動作を実現する方法として，ptherad を用いる．pthread は，本研究で対象とする，組み込みシステムに存在するタスクを，PC 上で実現するための方法として利用する．pthread はメモリ空間を，一つのアプリケーションとして共有することから，fork を用いるよりも，タスクの概念に近いことから選択した．

またセマフォの確認項目において，”メッセージを呼び出したオブジェクトを取得する”，とする項目がある．しかし AspectC++では JoinPoint としてフックさせるメッセージの呼び出し元を取得することが出来ない．そのため，around 句を用いて，呼び出し元のスレッド ID を取得することで同様の動作を実現させ，セマフォの確認をとることが出来るかという視点にて評価を行った．

AspectC++プログラム上で”around”とは，その JoinPoint であるメッセージに到達すると，around 以下に記述された内容を実行することを意味している．はじめに around 句では，セマフォを取得する”get_sem()”が呼び出された時点で，そのスレッド ID を取得する．その後 JoinPoin API の”proceed()”を用いて，通常の処理に戻している．set() や get(), free_sem() も同様にスレッド ID を取得している．

確認のながれとしてはじめに，共有資源にアクセスする時点で，その共有資源へのアクセスするスレッドと，セマフォを取得しているスレッド ID が同一であるかを確認をする．次にセマフォが解放された時点で，セマフォを解放したスレッド ID と共有資源へアクセスしたスレッド ID と，セマフォを取得しているスレッド ID が同一であるかを確認する．以上の手順によりセマフォを確認する．

```
aspect SemafoCheck{
    pthread_t pth, getth, freeth, tmpgetth, tmpfreeth;

    advice call("% get_sem()") : around(){
        tmpgetth = pthread_self();
        tjp->proceed();
    }

    advice call("% get_sem()") : after(){
        if(*tjp->result()){
            getth = tmpgetth;
        }
    }

    advice call("% set(%)") : around(){
        pth = pthread_self();
        tjp->proceed();
    }

    advice call("% get()") : around(){
        pth = pthread_self();
        tjp->proceed();
    }

    advice call("% free_sem()") : around(){
        tmpfreeth = pthread_self();
        tjp->proceed();
    }

    advice call("% free_sem()") : after(){
        if(*tjp->result()){
            freeth = tmpfreeth;
            if((freeth == pth) && (getth == pth)){
                printf(" >> semafo OK!\");
            }
        }
    }
};
```

第5章　事例の評価

5.1　概要

本章では事例に対し，設計制約確認ツールを用いて評価を行った結果について述べる．はじめに事例について整理し，その後行った評価について述べる．最後にその評価結果をまとめる．

5.2　事例の説明

評価対象とする事例は実際に動作するカーオーディオをモデル化したものを使用した．カーオーディオのシステムは，企業により提供されたものである．また評価は対象をそのままの状態で評価を行わずモデル化を行った．理由としては，規模が大きすぎること，PC上でカーオーディオのシステムを実行することが出来ないこと，AspectC++のサポートしているGCCではコンパイルできないことなどが挙げられる．モデル化の指針としては，実際のカーオーディオの動作のうち基本的な動作をのみを抽出した．また抽出したモデルからPC上で動作するプログラムを作成し，作成したプログラムを評価の対象とした．

評価対象とする一連の具体的な動作は，”外部からの入力に対し，その入力値に対応した動作を行う過程”とした．対称とした動作は以下の手順にて行われる．またUMLにより作成された全体のクラス図を図5.1, 対象とする機能を構成するクラス図を図5.2，状態遷移図を図5.3に示す．

対象動作

1. 外部から入力される
2. 入力された値と，現在のシステムの状態から行うべきイベント情報を，イベントテーブルクラスから取得する
3. 取得したイベント情報に従い，イベントを各制御部に送信する
4. イベントを受信した各制御部は，取得したイベントから，行うべき動作を取得する．
5. 各制御部は，行うべき動作を行い，H/Wとのインターフェースに行う動作を送信する．

また，他の評価した動作については，評価結果の項にて後述する．

クラス図 –全体構成–

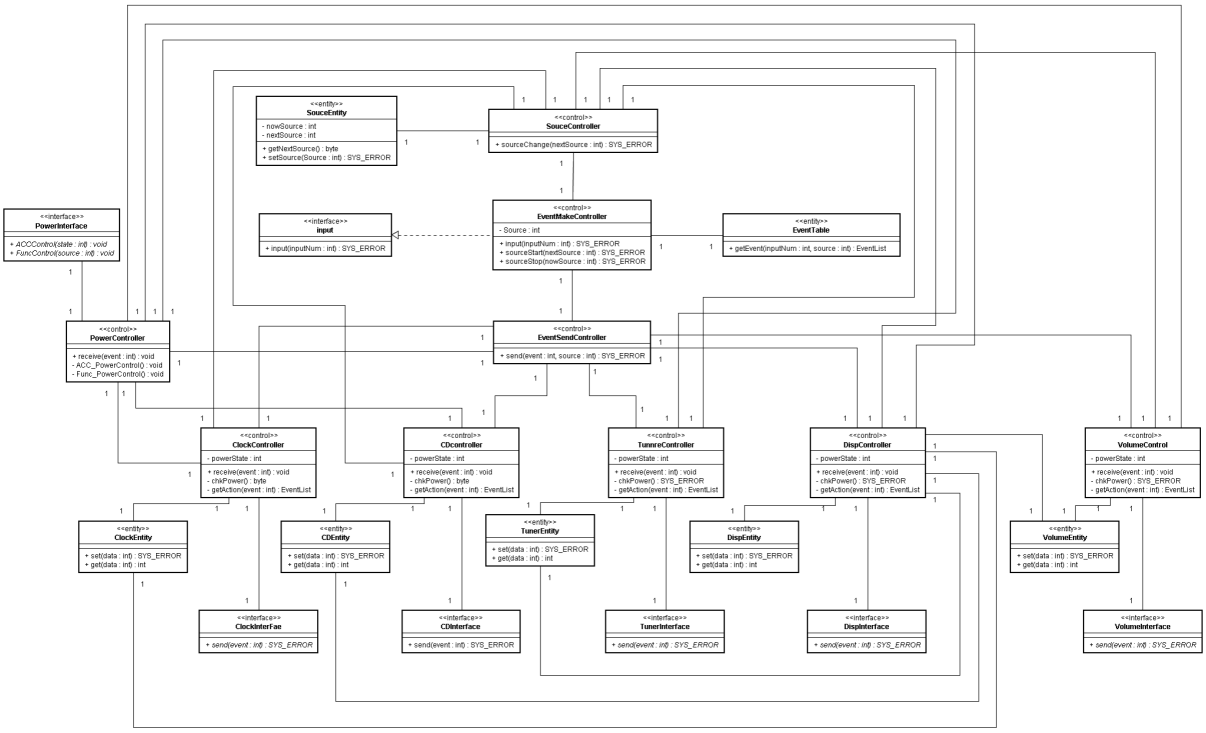


図 5.1: カーオーディオシステムクラス図

- イベント処理関係
  [**EventMakeController, EventSendController, EventTable**]
  入力値に対し，現在のソースから，動作するイベントを取得する．取得したイベントは適切な制御部に送信する．イベントはリスト形式であり，送信先と送信するイベントの組になっている．イベント毎に送信先が異なるために，リストの長さもイベントにより異なる．送信はイベントリストが無くなるまで続ける．

  イベント制御部は，どのような動作を行うべきかは判断せずに，入力値とソースに対応したイベントを送信する，ことを行う．

  また，input インターフェースクラスと実現の関係であるため，実際の input() 関数はイベント制御部にある．

- CD 制御
  [**CDContorller, CDEntity, CDInterface**]
  CD を制御する．再生や早送り，停止などのイベントを取得し，制御する動作をインターフェースに送信し，動作状態を CD 情報に設定をする．CD 制御部は取得したイベントから，さらにどのような制御を行うべきかのテーブルを自身で持っている．

- チューナ制御
  [**TunerController, TunerEntity, TunerInterface**]
  チューナを制御する．局選択などのイベントを取得し，制御する動作をインターフェースに送信し，動作状態状態をチューナ情報に設定する．基本動作はCD制御とほぼ同様である．

- 表示制御
  [**DispController, DispEntity, DispInterface**]
  表示部を制御する．どこの情報が更新されるかのイベントを取得する．更新される情報を取得し，動作状態情報を表示情報として設定し，表示更新するためのイベントをインターフェースに送信を行う．表示制御も基本動作は，CD制御やチューナ制御と同様である．

- 音量制御
  [**VolumeController, VolumeEntity, VolumeInterface**]
  音量を制御する．音量を上げる，下げるというイベントを取得し，制御する動作をインターフェースに送信し，動作状態状態を音量情報に設定する．基本動作はCD制御やチューナ制御と同様である．

- 時計制御
  [**ClockController, ClockEntity, ClockInterface**]
  時計を制御する．実際のシステムにおいては内部でクロック監視を行い，時間の変化とともに，表示制御部に更新を行うためのイベントを送信するが，今回の評価においては省略した．

- 電源制御
  [**PowerController, PowerInterface**]
  電源を制御する．起動と同時に各制御部を起動する．またソース切り替え時においては，制御部を一時的に電源オフのような状態にすることも行う．

- ソース制御
  [**SourceController, SourceEntity**]
  ソース切り替え制御を行う．現在のソース情報や切り替える対象のソース情報などを管理する．イベント制御部からソース切り替えのイベントを取得し，現在のソース情報，次のソース情報から，ソースを切り替えるイベントを再度，イベント制御部に送信する．またしばらく待ち，確実にソースが切り替わっているかを直接各制御部に確認も行う．

クラス図 –イベント送信–

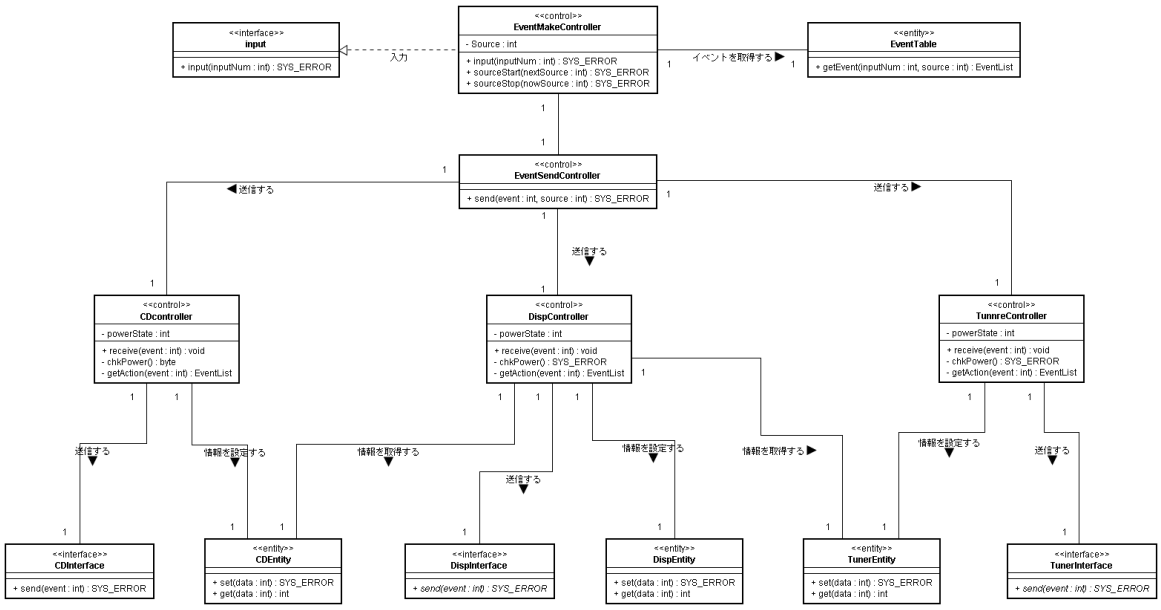
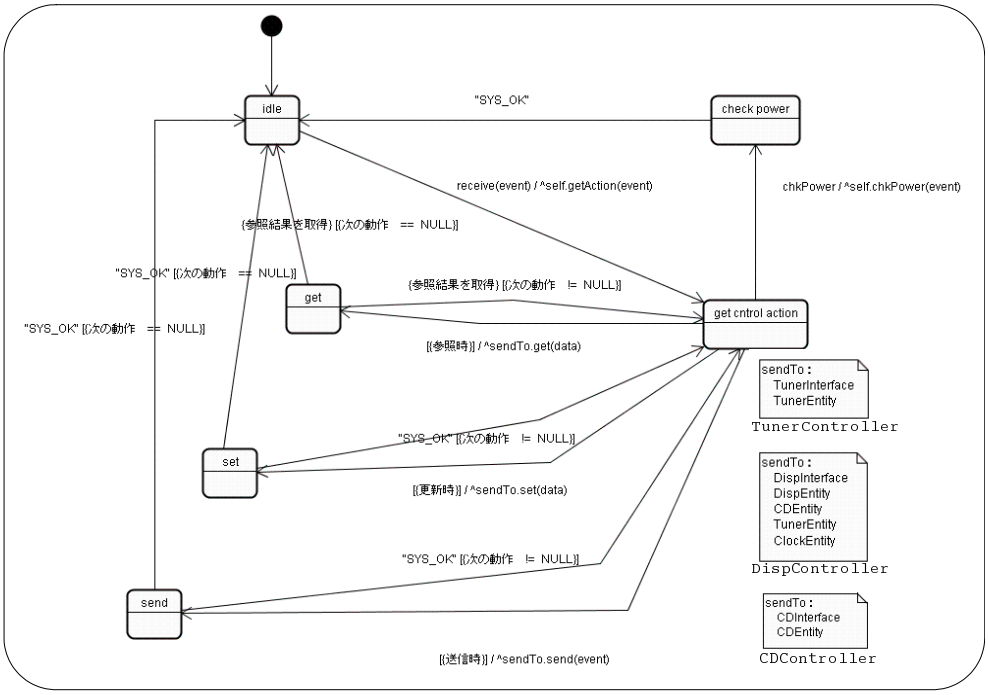


図 5.2: イベント送信機能クラス図

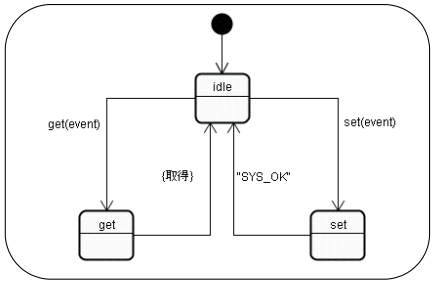
今回評価対象とした機能は，”**入力に対して，適切なイベントを送信し，適切な動作を各制御部が行う**”である．評価対象とした機能を構成するクラス図を 5.2 に示す．具体的な動作例は以下の通りである．

1. CD ソース時に，CD 関係の操作である play ボタンを押され，play ボタンに対応した値が送信される．
2. 現在のソースと，入力値からイベントリストを **EventTbale** から取得する．
3. イベントリスト内にある送信先に，対応するイベントを送信する．**CDController** に”**再生する**”が送られ，**DispController** に，”**CD 表示更新**”を送信する
4. **CDController** は **CDInterfce** に送信し，**CDEntity** に情報を設定する．
5. **DispController** は **CDEntity** から情報を取得し，**DispInterface** に送信し，**DispEntity** に取得した情報を設定する．

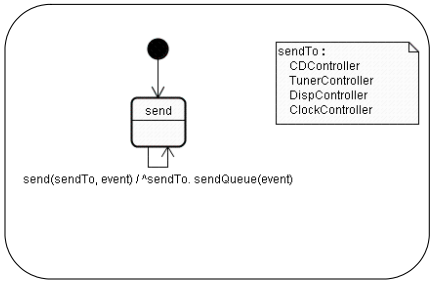
状態遷移図



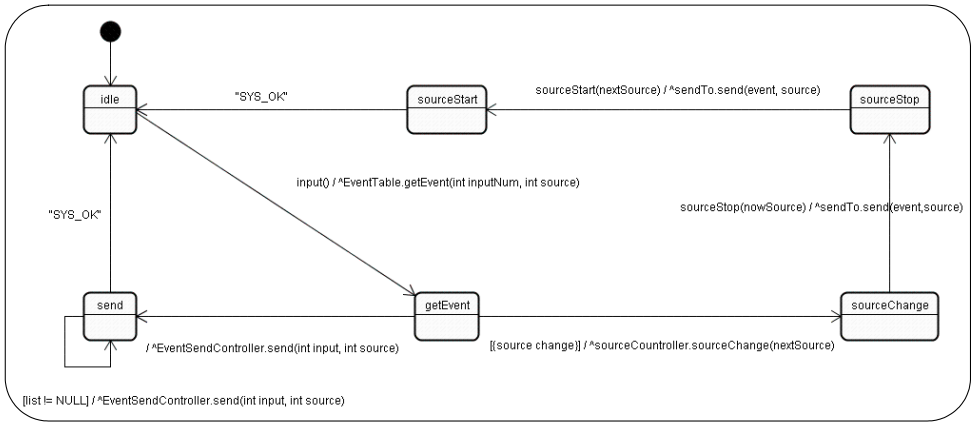
CD/Tuner/Disp Controller class



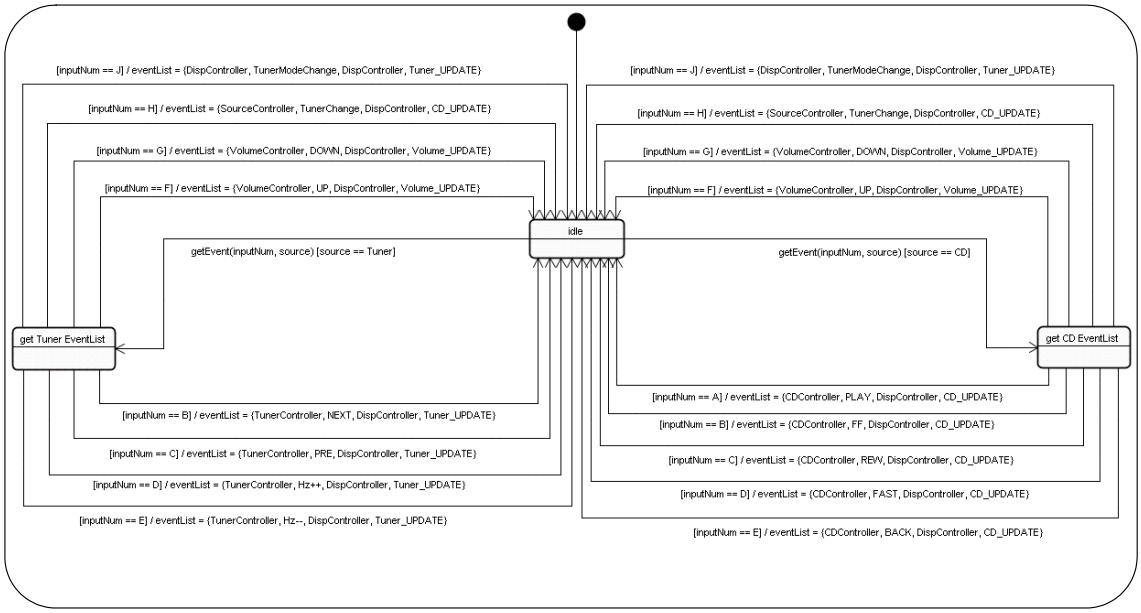
CD/Tuner/Disp Entity class



EventSendController class



EventMakeController class



EventTable class

図 5.3: イベント送信機能の各クラスの状態遷移図

1. input インターフェースクラスの実現をしている **EventMakeController** クラスに外部より入力される．

2. **EventMakeController** クラスは入力された値と，現在のソース情報から，行うべき動作を **EventTable** クラスから **EventList** として取得し，**EventSendController** クラスに **EventList** を渡す．**EventList** には，送信すべきクラスと，イベントがペアになった形で存在する．

3. **EventSendController** クラスは，**EventList** にしたがい，リストにあるイベントを送信し続ける．イベントはキューに入れらる．

4. 各制御部はイベントキューにイベントが入り次第，イベントの処理をする．**CDController** クラス，**TunreController** クラスは取得したイベントを処理するために，**各 Interface** クラスに送信する．送信が成功することを確認し，**各 Entity** クラスに現在の状態を設定する．

5. **DispController** クラスは，イベントキューにイベントが入り次第，イベントの処理をする．取得する制御部の **Entity** クラスから情報が更新され次第，情報を取得する．

更新されたことはポーリングにより監視を行うことで検知することが出来る．取得した情報を **DispInterface** クラスに送信し，成功することを確認し，**DispEntity** に情報を設定する．

5.3 事例を用いた確認方法とツールの評価

5.3.1 評価する性質

事例に対し，評価する性質は以下の 3 つである．

1. 処理の順序
   処理の順序の確認では入力から，イベント取得，イベント送信，各制御部での処理を確認する．処理の順序を確認するためのシーケンス図は5.4 となる．図中，**CDget()** 関数が **loop** フレームとなっているのは，ポーリングによるデータ監視のため，情報が更新されるまでアクセスをするために **loop** フレームとした．

2. データフロー
   データフローの確認は，変数の確認方法を用いる．**EventSendControler** クラスから送信されたイベントが，**CDController** クラスで取得され，情報として **CDEntity** クラスに設定される．さらに，**CDEntity** クラスにて設定された値を **DispController** クラスが取得し，設定しているかを確認する．これにより，**EventSendController ⇒ CDController ⇒ CDEntity ⇒ DispController ⇒ DispEntity** と行われるデータフローを確認する．

3. セマフォ
   セマフォの確認では **CDEntity** への情報の設定・取得は CD タスクと Display タスクから行われる．そこで共有資源である **CDEntity** へのアクセスへの，セマフォが守られているかを確認する．

データフローの確認のための，変数の確認には以下のように OCL として記述される．(入力値が”10”の場合，送信される値は”1”である)

CDController クラスへの送信イベント値を確認

```
context :: EventSendController:setQueue(int:event)
    pre : event = 1;
```

CDEntity クラスへの送信値を確認

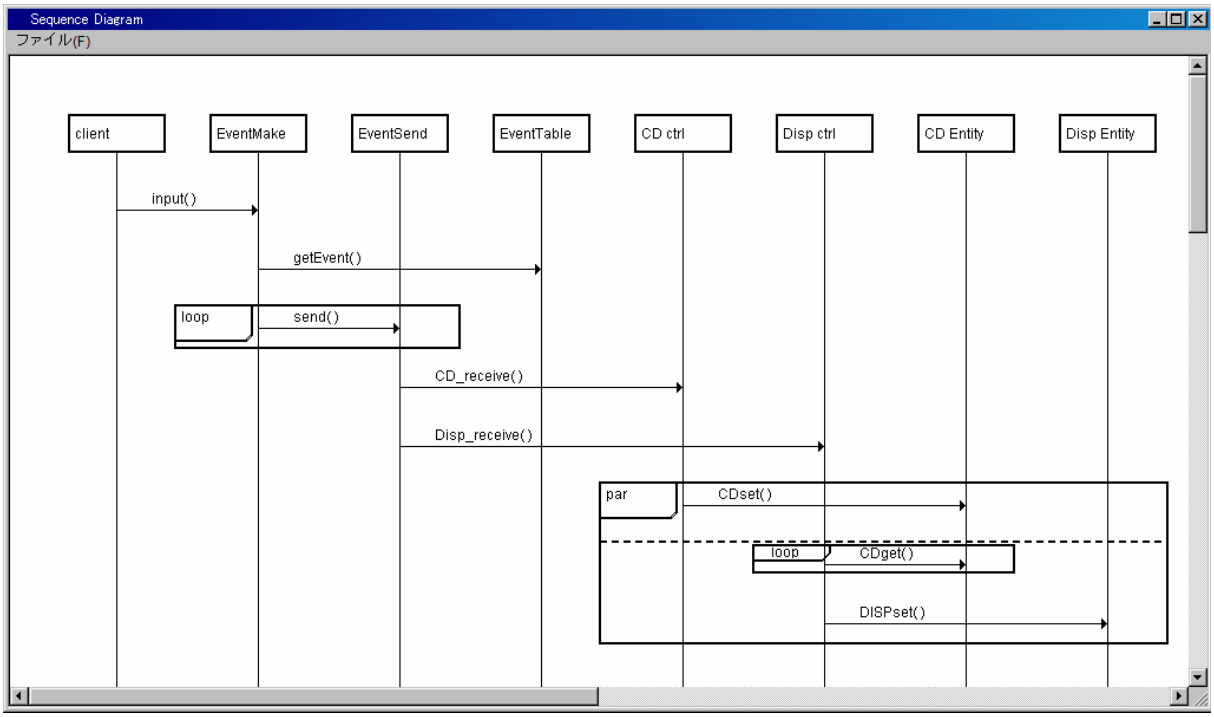


図 5.4: 処理の順序確認のシーケンス図

> **context :: CDEntity:CDset(int:data)**
> **pre : data = 1;**

CDEntity クラスから取得するベント値を確認

> **context :: CDEntity:CDget()**
> **post : result = 1;**

DispEntity クラスに設定するイベント値を確認

> **context :: DispEntity:DISPset(int:data)**
> **pre : data = 1;**

セマフォの確認

セマフォの確認には以下のように OCL として記述される.
CDEntity クラスに情報を設定する.

context :: CDEntity:CDset()
pre:CDEntity.sem_get().getCallObj()=CDEntity.CDset().getCallObj()
post:CDEntity.CDset().getCallObj()=CDEntity.sem_free().getCallObj()

CDEntity クラスから情報を取得する.

context :: CDEntity:CDget()
pre:CDEntity.sem_get().getCallObj()=CDEntity.CDget().getCallObj()
post:CDEntity.CDget().getCallObj()=CDEntity.sem_free().getCallObj()

5.3.2 評価結果

その他，様々な状況下でも設計制約が守られていることを確認した．評価した例は，以下のような動作である．

- イベントを送信し，各制御系にて処理を行う
  処理の順序や，排他性，データフローの確認
  - CD イベント (再生，早送り，巻き戻し)
  - チューナイベント (チャンネルの変更)
  - 音量イベント (音量を上げる，下げる)
- ソースを切り替える
  - CD からチューナへの変更
  - チューナから CD への変更

事例を評価した結果，処理の順序，変数の確認によるデータフロー，セマフォの確認をすることが出来た.

今回は，自分で実際にあるカーオーディオシステムに対してモデル化を行い，実装したプログラムであったために，事例の評価によりプログラムの誤りを見つけることが出来た.

データフローの確認おいて，**DispEntity** クラスに取得した情報を設定する動作にて作成したプログラムに誤りを見つけた．**CDEntity** から情報を取得する際に，値の処理において違う変数の値を使用するという誤りであった．この誤りは **DispEntity** クラスへ設定する値を取得時に誤りを見つけるた.

また，事例におけるセマフォの確認が出来た．今回の事例においてセマフォの確認は，スレッド ID を呼び出し側として取得し，比較，確認を行う．しかしスレッドを用いて実

装していない場合などには，スレッド ID を取得し，比較する方法を用いることが出来ない．他に何か取得する方法の検討が必要であると思われる．よって条件は限定されるが，スレッドを用いている場合においてはセマフォの確認に有効であることが得られた．

以下は実行結果の一部である．出力中に，変数の確認の結果が **OCL CHECK OK!**と出力されている．またセマフォの確認の結果が **semafo OK!**と出力される．最終的に処理の順番の確認結果が積まれたスタックと，確認した正規表現，確認結果が出力される．

```
INPUT NUMBER! : 10

send is called!
        >> OCL CHECK OK!
        >> [event==1]
        >>send is called!
        input ->
                getEvent ->
                        send ->
                        send ->

        << ··· 省略 ··· >>

        >>CD sem free! :from–5
        >> semafo OK!
        >>CD sem get! :from–1
        >>CDset is called!
        input ->
                getEvent ->
                        send ->
                        send ->
                                CD_receive ->
                                        Disp_receive ->
                                                        CDget ->
                                                        CDget ->
                                                        CDget ->
                                                        CDget ->
                                                        CDget ->
                                                        CDget ->
                                                CDset ->
        >> CHECK par01
        >> MATCHED
        match method : f
        >> 元の文字列 : f
        >> match した文字列 : f
        >> CHECK OK!
        >> OCL CHECK OK!
        >> [data==1]

                << ··· 省略 ··· >>

CD Entity get data: 1
        >> OCL CHECK OK!
        >> [result==1]
```

```
>>CD sem free! :from–5
>> semafo OK!
>>DISPset is called!
input ->
        getEvent ->
                send ->
                send ->
                        CD_receive ->
                                Disp_receive ->
                                                CDget ->
                                                CDget ->
                                                CDget ->
                                                CDget ->
                                                CDget ->
                                                CDget ->
                                        CDset ->
                                                CDget ->
                                                        DISPset ->
>> CHECK par02
>> MATCHED
match method : g*h
>> 元の文字列 : ggggggh
>> match した文字列 : ggggggh
>> CHECK OK!
>> CHECK strict
>> MATCHED
match method : abc*de0
>> 元の文字列 : abccde0
>> match した文字列 : abccde0
>> CHECK OK!
>> OCL CHECK OK!
>>[data==1]
```

第6章　考察と今後の課題

6.1　概要

アスペクト指向言語を使用した設計制約の確認について考察を行う．設計制約を確認する際に，本研究にて提案する方法を用いて確認できること，出来ないことや，確認する上での問題点，評価結果から得られたことなどについてまとめる．

本章では，評価を行った際に，アスペクト指向言語を用いることの問題点と，正規表現による確認方法の問題点があることがわかったために，この 2 点を中心に整理を行う．

6.2　アスペクト指向言語を使用する問題点について

本項ではアスペクト指向言語を用いる上での問題点について考察を行う．本研究に用いる AspectC++は，目印となるプログラム上の特徴点に対して，アスペクトで記述される機能を追加することができる言語である．しかし，目印となる特徴点や，機能の追加方法などで AspectC+でできることには限界があるために，それらの影響による問題点について整理を行う．

6.2.1　実行時間に関する問題

Sleep() 関数などの実際の実行時間に大きく関係する動作では，通常よりも実行時間のかかる AspectC++プログラムをウィーブする際に，Sleep() の及ぼす範囲が大きく変わってくるために注意が必要である．例えば，ある処理が終了するまでの処理時間が 5 秒であるときに，他の処理が Sleep 関数により，”5 秒間”と指定して待機するような状況があったとする．このとき，ある処理にアスペクトで非常に時間のかかる処理をウィーブした場合には，ある処理の処理時間が 5 秒以上になり，Sleep 関数を使用して待っている他の関数に影響を与えてしまうことがある．また実行時間の問題は，クロス環境とセルフ環境との間で大きく異なってしまうために，非常に難しい問題でもある．そのため，実行時間に関係する処理を行う場合には，アスペクトをウィーブすることで生じる，アスペクトの処理時間を考慮しなくてはいけない．

6.2.2　メッセージのフック方法の問題点

AspectC++は基本的に，関数の呼び出しやコンストラクタなどの，プログラマーにとって意味のある箇所を，JoinPpoint としてフックさせる．しかしそのフックさせることの出来る箇所であるプログラム中で関数の呼び出しなどは，様々な箇所で，様々な方法により使用される．しかし，AspectC++では呼び出す意味上の判別は行わず，すべて同じようにフックさせてしまう・その結果，確認したいメッセージが，自分の想定外の箇所で使用される場合は，本研究による方法では確認時に誤動作することがある．

例えば，図 6.1 のような場合を考える．点線で囲まれた部分が実際には存在するが，確認の際に特に重要ではないと判断され，シーケンス図に記述しなかったメッセージであるとする．この時シーケンス図に記述され，処理の順序を確認するための正規表現は”abc”で表現されるが，実際に確認される ID 列は”abac”となる．このように，対象とするメッセージは，すべてシーケンス図に記述しなくてはいけない，という問題がある．この問題を解決する方法としては，フックさせる条件を条件を増やすことで，この問題を回避することが出来ると思われる．例えば，メッセージを呼び出すオブジェクトの指定を，AspectC++では”within”を用いることで限定することが出来る．また，呼び出す際の引数の値などでも，限定していき，精度を上げられることが期待できる．

現状では，シーケンス図に記述するメッセージは，その確認対象とする範囲内にて呼び出される箇所を全て，シーケンス図に記述する必要がある．

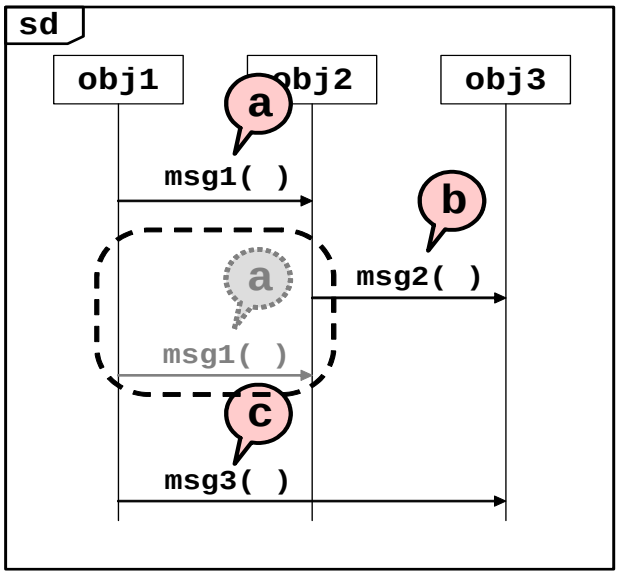


図 6.1: 処理の順序の確認時に，エラーが発生する場合

6.2.3 正規表現による確認方法の問題点

正規表現による確認の問題点は，繰り返しを正規表現にて確認する点にある．そのため確認時に誤判定しないよう範囲を明確にする必要がある．この問題も，メッセージのフックの方法と関係している問題でもある．フックさせる際に，より詳細な条件で限定をかけることで，回避できる点があると思われる．

繰り返しによる問題とは，例えば一つの msg() というメッセージが 10 回繰り返す状況がある場合を図 6.2 に示す．msg() の ID を仮に”b”とし，正規表現を”b ＊”とした時，以下のような振る舞いの時にマッチし，確認ができたと判定される．

- 1 回も実行されない
- 5 回繰り返す
- 20 回繰り返す ･･･ など

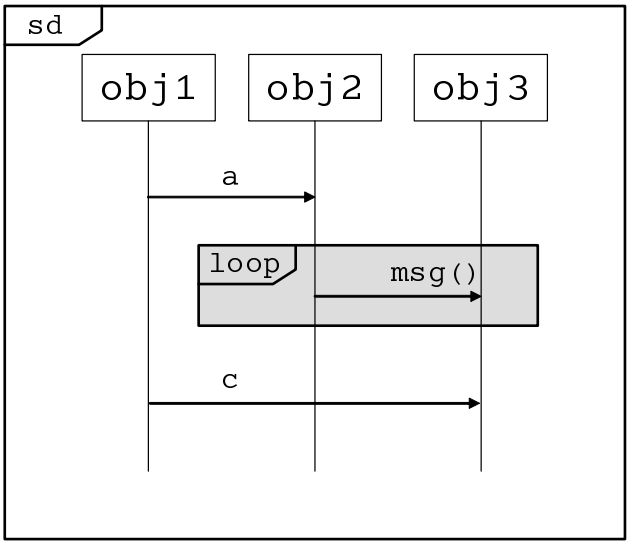


図 6.2: 繰り返しが起こる例

この状況では，上記の 3 つ全て確認できたと判定してしまう．そのために，繰り返し条件にて確認できる範囲というものを決める必要がある．以下にその例を挙げる．

- ”＊”を使用するとき，0 回以上の繰り返し
- ”＋”を使用するとき，1 回以上の繰り返し

- 繰り返しの回数は確認しない

また図 6.3 のような場合，loop フレームの後のメッセージが行われているかの確認は出来ない．つまり

$$\mathbf{loop[msg() \Rightarrow msg() \Rightarrow msg() \Rightarrow msg()] \Rightarrow msg()} \quad [1]$$

となる場合，正規表現による確認では，確認できることがあいまいになってしまう．図 6.3 ような状況下では確認できる範囲は，loop フレームとその直後のメッセージを含み，正規表現でマッチする範囲が確認できる．

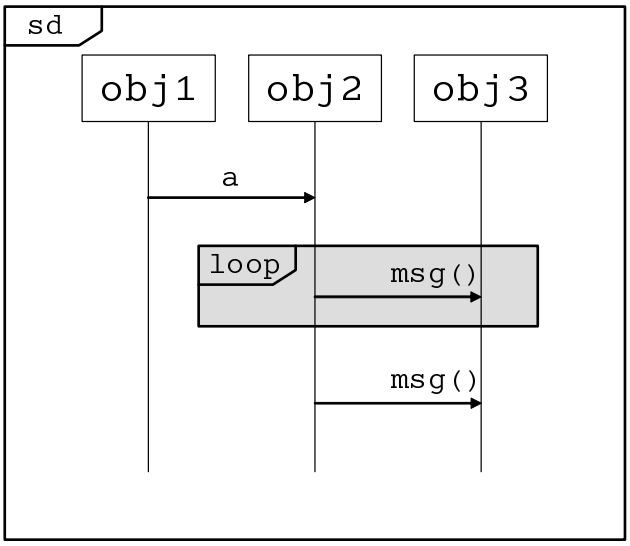


図 6.3: 繰り返しと識別できない例

本研究にて UML2.0 のシーケンス図にて使用されるフレームのうち，sd(strict), alt, loop, par, seq を使用した．しかし他にも様々な表現を行えるフレームが存在しているため，他のフレームを用いて，さらに表現，確認できる項目がないか検討を行うことが出来ると思われる．たとえば，”neg”という，「”neg”フレーム内の関係には問題がある」という否定を表す表現がある．”neg”フレームを用いることで，存在してはいけないメッセージを確認することが出来るなどが期待される．

[1] ここで，loop[···] の [ ] 内は，フレーム内のメッセージを表現している．

6.3 OCL を用いた変数の確認について

変数の確認は，データフローを確認するための方法として用いている．しかし本研究にて使用した AspectC++では，変数への代入を JoinPoint とすることが出来ないために，確認対象は JoinPoint となる関数の引数，返り値のみであった．しかし AspectJ など他のアスペクト指向言語では JoinPoint に変数とすることも可能なため，より詳細なデータフローを確認することが出来ると思われる．

また，変数の確認では対象とするメッセージが呼ばれた時，と限定していた．しかし，より強い限定をつけ，同じメッセージが複数回呼ばれるときの順番まで指定できるようにすると，処理の過程で順次，値が変化するデータフローなども確認できるようになると思われる．例えば，あるメッセージが複数回呼ばれる場合，3 回目に呼ばれたときの変数の値に条件をつける，などの条件である．

6.3.1 セマフォの確認方法について

シーケンス図と OCL を用いた確認において OCL を拡張し，セマフォを確認する際の問題点について考察をする．セマフォの確認は，今回は実装言語に依存した確認方法となった．しかし，実行環境，実装言語に依存せずにその問題を解決することは，現段階では非常に困難であるが，もう少し一般的な確認方法について検討を行う必要がある．

第7章　関連技術

7.1　JBoss AOP

AspectJ に並ぶ代表的なアスペクト指向言語である．JBoss AOP は java プログラムの形でアドバイスを定義する．一方 pointcut などの情報は別途 XML を使って”jboss-aop.xml”という定義ファイルに記述する．AspectJ や AspectC++ではコンパイル時にウィーブするのに対し，JBoss AOP ではクラスのロード時に動的にウィーブされ，このように動的にウィーブすることを，”Hotswap Weaving”と呼ばれる．

AspectJ では”advice”や”pointcut”を定義するために、独自のキーワード”aspect”を用いた「アスペクト」を作成したが、 JBoss AOP では「インターセプタ」と呼ばれるクラスを使用する．AspectJ のように Java の拡張仕様を用いるのではなく，通常の Java のクラスを用いてアスペクトの振る舞いを記述する．

JBoss AOP は記述スタイルとしては，Advice に相当する実際の処理は Java で実装され，PointCut は，別の XML ファイルに記述されるのが特徴である．そのため，JBoss AOP では実装形態は通常の Java プログラムと同じでとなる．一方 AspectJ や AspectC++では，ウィーブの対象言語の拡張で行うために，ウィーブ先で Aspect 言語に対応しないような機能を使用された場合には，ウィーブ出来なくなる場合がある．JBoss AOP では，外から見た場合にはただの Java アプリケーションでしかないため，対象範囲が広くなる．一方，一つのアスペクト定義に対し，2 つの以上のファイルで管理する状況が考えられるため，保守の面で注意する必要があると思われる．

また機能の面は，AspectC++よりも高機能で，様々な JoinPoint をとることができる上に，Javade できる事は何でも使用可能である．特に AspectC++では取得できなかった変数への値の代入も，JoinPoint とすることが出来る．変数の代入を JoinPoint と出来るため，本研究にて提案するデータフローの確認の精度を上げることが可能と思われる．

7.2　AspectJ

AspectJ は，米ゼロックス社の PARC 研究所の Gregor Kiczales によって Java のアスペクト指向言語として開発された．

言語仕様としては，本研究で使用した AspectC++のモデルとなった言語でもあり，AspectC++と AspectJ は非常に似た言語になっている．ウィーブする箇所を JoinPoint と

し，その集合を PointCut ととし，アスペクトとして記述されるモジュールを，advice とするなど表現の仕方などに多少の相違点は見られるものの，ほぼ同じである．表現の違いとしては，引数や返り値として置き換えることの出来る文字は，AspectJ では”＊”だが，AspectC++では”%”である．これは C/C++では”＊”をポインタとして使用するために他の文字として置き換える必要があったためである．そのほか JoinPoint API なども基本的な部分において同様の機能を持っている言語体系である．

AspectJ と AsapectC++の違いとして，AspectJ はコンパイルも行い，VM 上で動作するためのバイトコードに変換されるが，AspectC++では C コンパイラのプリプロセッサであり，C コンパイラでコンパイルできる C プログラムを作成し，通常の C コンパイラでコンパイルを行うといった違いがある．また機能の違いとして，変数への代入に対して JoinPoint と出来る点である．AspectC++では，変数の代入を JoinPoint とすることがきなかった．変数への代入を JoinPoint とすることができる AspectJ の場合，本研究にて提案するデータフローの確認の精度を上げることが可能と思われる．AspectC++では，引数と返り値のみであったが，AspectJ では，引数と返り値に加え，通常の変数も対象とすることが出来る．

7.3 Hyper/J

サブジェクト指向プログラミング，多元的な関心事の分離の考え方からの発展した Java のアスペクト指向言語である．普通の Java コンパイラでコンパイルしたバイトコードから,「ある観点」で hyper slice を抜きだし，それを他のアプリケーションに追加することができる．この「ある観点」がサブジェクト指向から発展した観点であり，この hyper slice が concern と呼ばれる関心事に相当する．hyper slice とは，concern に関連するメソッドと変数のみを含むクラスから構成されるモジュール単位である．

AspectJ においてはクラス構造が前提であるが，Hyper/J ではクラス構造は持たなく，より一般的な概念でアスペクトとして捉えることが出来る．しかしその一方言語体系は複雑になる．

他のアスペクト言語と大きく違う点は，あらかじめ concern を分離して記述するのではなく，記述されてあるアプリケーションのバイトコードから， concern を抜き出して再利用できる点である．

Hyper/J では，AspectC++や AspectJ，JBOSS AOP に比べ，横断的な関心事として捉える対象が，より一般的であり，広い概念になる．例えば，AspectC++ではクラス構造が前提となった考え方であるが，Hyper/J ではクラス構造を持たないで使用できる．そのために，横断的な関心事を的確に捉えることが出来れば，本研究によって提案する方法よりも確認できる対象が大きくなることが考えられる．しかし，横断的関心事としてプログラムレベルではない捉え方となるために，確認方法が複雑でになることも考えられる．

第8章　終わりに

8.1　謝辞

本論文を執筆するに当たり終始ご指導賜りました，片山卓也教授，岸知二客員教授，青木利晃助手に感謝申し上げます．また本研究に対してご意見を頂いたり，質問や議論にも快く応じて下さいました片山研究室，岸研究室，デファゴ研究室の皆さんに感謝いたします．

参考文献

[1] *Olaf Spinczyk, Andreas Gal, Wolfgang Scroder-Preikschat, AspectC++: An Aspect-Oriented Extension to the C++ Programming Language, 2002*

[2] *Bart Verheecke, Ragnhild Van Der Straeten, Specifying and Implementing the Operational Use of Constraints in Object-Oriented Applications, 2003*

[3] 岸 知二, 野田 夏子, ソフトウェアアーキテクチャ, コンピュータ ソフトウェア *Vol.18, 2001*

[4] *UML 2.0 Superstructure Specification, Object Management Grouop, 2005*

[5] *UML2.0 OCL Specification Object Management Group, 2003*

[6] *Matthias Urban, Olaf Spinczyk, ApectC++ Language Reference, 2004*

[7] *J* ヴァルメル, 竹村 司, *UML/MDA* のためのオブジェクト制約言語 *OCL*, 星雲社, *2004*