修 士 論 文

ソフトウェアパターンに基づく体系的設計・検証手法の研究

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

金井勇人

2006 年 3 月

修 士 論 文

ソフトウェアパターンに基づく体系的設計・検証手法の研究

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

410031 金井勇人

提出年月: 2006 年 2 月

概 要

本研究ではソフトウェアの構造と性質を合わせて，パターン化する手法を提案する．ソフトウェアのそれぞれの構造によって確認したい性質や、その性質の確認方法にはいくつかの定石があり、それを体系だって提示することが有効であると考える．また，仕様記述言語の記述と検証したい性質を記述する論理的な記述はとても関連しているので合わせて，支援する必要がある．

本研究では，上記で提案した手法より，検証者の形式的記述の支援を行うことができた．

目 次

第1章　はじめに

近年，組み込みソフトウェアの開発・検証手法は従来のものでは十分に対応しきれなくなっている．その背景として，色々なところで組み込みソフトウェアが使用されるようになり，その信頼性が社会的な問題となっていることが挙げられる．また組み込みソフトウェアは大規模，かつ複雑になってきており，従来の開発手法の限界が指摘されている．よって，経験測による手法だけでなく，科学的手法の導入が検討されている．

形式的検証手法の１つにモデル検査 [3] がある．この検証技術は，ソフトウェアの有限状態モデルが，論理で表現された性質を満たすかどうかを状態の網羅的な探索によって検証を行う技術のことである．モデル検査技術を UML 等で記述される設計モデルに適用することにより，ソフトウェアの信頼性を高めることが期待できる．しかし，UML 等で記述されたソフトウェアのモデルをモデル検査ツールで検証するために，そのツールに依存した言語 (以下，仕様記述言語と呼ぶ) で記述しなければならない．また，モデル検査ではソフトウェアが満たしてほしい性質を確認するために，性質を時間的な概念を持たせた論理式 (以下，時相論理式と呼ぶ) で記述することも必要になる．この時相論理式は仕様記述言語に対応した形で記述する必要がある．こうした作業はそれぞれの文法をよく理解し，ソフトウェア検証に対応したテクニック的な記述を考える必要があり，ソフトウェア技術者にとって，大変な作業となる．その上，検証したい性質が複雑になると，時相論理式が非常に困難になる．

ソフトウェアのそれぞれの構造によって確認したい性質や、その性質の確認方法にはいくつかの定石があり、それを体系だって提示することが有効であると考える．そのため，仕様記述言語の記述と検証したい性質を記述する論理的な記述を合わせて，支援する必要がある．

本研究ではソフトウェアの設計モデルに対しての形式的検証を行う際の形式的記述を支援することを目的とする．具体的には，ソフトウェア検証の定石をソフトウェア技術者が利用しやすい形でパターンとして提示する．詳しくは第 2 章で説明する．

本論文の構成は以下の通りである．第 2 章では本論文での目的を述べる．第 3 章ではモデル検査とそのツールの一つである SPIN[1][2] の概要について説明する．第 4 章では構造付き検証パターンの提案を行う．従来の代表的な検証パターンとの比較も行う．第 5 章では事例を使って本検証パターンの評価を行う．第 6 章で本論文を総括する．

第2章　目的

2.1　解決したい問題点

本研究はUMLをモデル検査で検証する際の以下の２つの問題点を扱う.

- UML上の性質を毎回アドホックに時相論理式に変換するのは大変である.
- UMLから仕様記述言語への変換は自明ではない.

上記２つの問題点は密接に関係している．以下に本研究の対象としているモデル検査ツールSPINの仕様記述言語であるPROMELAと時相論理式であるLTLの例を示す.

PROMELA

以下にPROMELAの例を示す.

```
proctype A(){
 ...
 L1: ch!MSG ->
 ...
}

proctype B(){
 ...
 ch?MSG ->
L1:    skip;
 ...
}
```

LTL

以下にLTLの例を示す.

```
#define p A@L1
#define q B@L1
```

```
[](p -> <>q)
```

上記の例は，プロセスAがプロセスBにメッセージを送っているPROMELAである．LTLは「プロセスAがメッセージを送信したら，必ずプロセスBがいつか受信する」という性質を記述している．LTLの記述はPROMELA記述に依存するので，その両者を合わせて考えることが有効である．したがって，PROMELAをどう記述するかによって，LTLで記述した性質の意味が変わってしまう．

この関係を考慮して，両方の記述の支援を考える必要がある．

2.2 目的とアプローチ

1つ目の問題点である，時相論理式の記述に関する問題は，パターンを利用する方法が提案されている．しかし，このパターンは4.1節でも説明するが，時相論理式のみのパターンで，非常に汎用的で一般的である．2.1節で述べた例の性質は「応答性」という非常に一般的な性質としか，パターン化されない．これでは，UML上の性質の記述の支援としては不十分である．そこで，UML上の性質を使って具体的な構造を時相論理式とセットで，考えることによって具体的なソフトウェアの性質をパターン化し，それをSPINに適用することを目的の1つとする．

また，もう1つの問題点である，UMLの仕様記述言語への変換に関する問題は，2.1節で説明した，時相論理式との密接な関係があるので，それを考慮しながら本研究の対象であるPROMELAへの変換方法を提案する．具体的には，UML上の特有の性質，例えば，「メッセージの送信，受信」などをどうPROMELAで記述するか，それをどうLTLに関連付けていくかということである．これを本研究の目的の1つとする．

第3章　モデル検査技術

本研究ではモデル検査ツール SPIN を扱う.

3.1　モデル検査概要

近年、複雑なシステムが多くなるにつれて自動検証技術への期待が高まってきている. 自動検証に多く用いられているのがモデル検査ツールで、既にいくつかのモデル検査ツールが実用化されている. モデル検査ツールの利用手順は，モデルの抽象化，対象システムの作成，状態空間の網羅的な探索という順番で行われるのが一般的である. 網羅的な探索の際に不具合の検出を行う. ここでの不具合とはデッドロックや無限ループ，仕様の違反などのことを指す. モデル検査ツールによる検証は人手による検証よりはるかに網羅的に行えることから，設計中のシステムの誤りを早期に発見する技術としても注目されている.

3.2　SPIN 概要

SPIN (Simple Promela INterpreter) はソフトウェアやプロトコルのためのモデル検査ツールである. SPIN で検証するモデルは Promela (PROcess MEta LAnguage) と呼ばれる言語で記述される. ベル研究所にいた G.J.Holzmann が中心になって開発し公開しているモデル検査ツールである. 開発当初はコンピュータプロトコルの設計と検証を主な目的としていたが，最近では，ソフトウェア検証のための基本ツールとして使われるようになっている. Promela は並列動作を記述する C ライクな専用言語であり，複数のプロセスが並列に動作するシステムを記述することができる. 変数，配列，構造体といった基本的なデータ構造のほか，プロセス間のメッセージ通信のための機構を標準で備え，実際のプログラムを組む感覚で記述できるのが特徴である. Promela で記述してしまえばそのまま検証が可能であるので，書いたその場で検証が可能なのが大きな利点となっている. SPIN はデッドロックや飢餓状態といった並列プログラムにおける基本的な問題を発見できる他，対象となっているモデルが要求されている仕様を満たしているかどうか，といった事を検証することも可能である.

3.2.1 PROMELA

SPIN では，仕様記述言語 PROMELA による記述を入力として，自動検証を行う．以下に，PROMELA を記述するモデルの主な文法について記述する．

proctype と init

PROMELA ではプロセスを定義するために，proctype 宣言文を用いて行う．例えば以下のように記述する．

```
proctype Example(){
    int n;
    n=1;
}
```

上記の例は int 型の局所変数 n を持つプロセスを宣言している．このプロセス型の名前は Example である．プロセス本体はとで囲まれ，いくつかの命令文によって成り立っている．上記の場合，局所変数宣言と 1 つの代入文から成り立っている．

プロセス宣言文はプロセスの宣言を行うだけで，それだけでは実行されない．実行開始時には init 型のプロセスのみが実行される．init 型のプロセスとは，C 言語のプログラムにおける main() 関数と同じ意味合いのものである．init 型のプロセスで行うことは，一般的に，大域変数の初期化，メッセージチャネルの作成，プロセスの起動であることが多い．以下に例を記述する．上記の例で示したプロセス Example の起動と byte 型の大域変数 m の初期化は以下のように記述する．

```
byte m;

init{
    m=10;
    run Example();
}
```

また，1 つのプロセス複数起動する場合は以下のように複数回起動文を記述すればよい．

```
init{
    run Example();
    run Example();
}
```

init 型プロセスで起動されたプロセスは起動された時点から並行に動く．

変数と配列

PROMELA における変数，配列は，大域的に使われる場合と，プロセス内のみで使用される局所的に使われる場合があり，それぞれ，宣言する場所に依存する．変数，配列の型は bit,boot,byte,short,int,chan の６つである．最初の５つの型は，基本データ型と呼ばれ，一時に１つの値を保持する変数，配列の宣言に用いられる．chan 型は，メッセージチャネルを記述するために用いられる．メッセージチャネルについては後述する．

制御文

まずは if 文である．例えば以下のように記述できる．

```
int n;
 ...
if
::(n==0) -> operational_sentence1
::(n>10) -> operational_sentence2
::else -> operational_sentence3
fi;
```

上記の例は int 型の変数 n の値が 0 か，10 より大きいか，またはそれ以外かで処理が選択される例である．この選択肢にはそれぞれ前に::がついていて，そのうちの１つの系列だけが実行される．上記の例では１つの系列が相互排他的に実行されるが，下記のような場合はそうではない．

```
int n;
 ...
if
::(n==0) -> operational_sentence1
::(n<10) -> operational_sentence2
::else -> operational_sentence3
fi;
```

この例は 1 つ前の例とほとんど同じだが，2 番目の条件が変数が n が 10 より小さい時，という条件であり，変数 n の値が 0 であれば，1 番目の条件と 2 番目の条件が実行可能となる．この場合，対応する系列の１つがランダムに選択される．これを利用した非決定的な記述方法については後述する．全ての系列が実行不可能であれば，すくなくとも 1 つの系列が実行可能になるまで，ブロックされる．

次に do 文である．選択の論理的な拡張が反復である．

```
int n;
 ...
do
```

```
::(n==0) -> n=n+1;
::(n>=10) -> n=0;break;
do;
```

上記の例は int 型変数 n の値を 1 増加させ，また初期値 (値 0) に戻す反復プログラムである．ここでは，1 つの選択肢だけが実行される．１つの選択肢が実行されたら再び同じことが繰り返される．反復が終了する一般的な方法は，break 文を用いることである．上の例では，変数 n の値が 0 になった時ループから脱出できる．

メッセージチャネル

メッセージチャネルは，あるプロセスから別のプロセスへのデータ，メッセージ転送をモデル化するために用いられる．そのために予約語 chan を用いて，大域的に宣言する．以下に例を示す．

```
chan ch0 = [0] of {int}
```

上記の例はチャネル ch が int 型のメッセージを受け取ることができることを表している．[] 内の数字は蓄えることができるメッセージの数を表す．この例では [0] となっているので，メッセージを蓄えることができない．意味としては非同期通信になる．また，[] 内の数字 0 以上の数字にし，メッセージを蓄えるようにした場合の意味は，非同期通信になる．送信側がバッファーになっているチャネルにメッセージをどんどん送信し，受信側は送信側がメッセージを送信しているタイミングとは関係なく，メッセージを受信する．このチャネルのバッファーは FIFO である．また，チャネルの配列は以下のように記述できる．

```
chan ch1[5] = [0] of {int}
```

上記の例では，int 型のメッセージを受け付けるチャネル ch が５つ宣言される．次に，チャネルに渡されるメッセージが複数ある場合，以下のように記述する．

```
chan ch2 = [0] of {int,bool,byte}
```

ここで，各メッセージは４つのフィールドから成り立っており，各フィールドはそれぞれ 32bit 値，1bit 値，8bit 値を表している．次の文は先程作ったチャネル ch0 に式 send の値を送ることを表している．

```
ch1!send
```

つまり，チャネルの終端に Message の値が付加される．

```
ch1!receive
```

上記の例はチャネルの始端からメッセージを取り出し，それを変数 receive に蓄える．複雑なメッセージを受け渡されるときにはコンマで区切ったり，括弧でくくったものを並べて表記する．また PROMELA では受信動作に関して，？の後に定数をおく場合は条件文として使うことができる．以下に例を示す．

```
 ...
int n=5;
ch2!n;
 ...
ch2?5 -> operational_sentence;
 ...
```

上記の例は，？の後に定数 5 が置いてあって，チャネルから 5 が受信できた場合だけ次の処理に進む．

3.2.2 LTL

時相論理式

時相論理とは，通常の論理式の構成要素である原始命題，論理積，論理和，論理否定の組み合わせに時間的な概念を持った時相演算子を加えたものであり，時間に関する概念を記述するのみ適している．LTL は時相論理式の 1 つである．原始命題とは事実を述べた命題であり，推論をすることなくその真偽を決めることができるものをいう．

LTL 式の構成要素

LTL 式は次の構成要素の組み合わせで表現できる．

- Strong Until

  $\sigma \models (pUq) \Leftrightarrow \sigma_i \models q \vee (\sigma_i \models p \wedge \sigma[i+1]) \models (pUq)$

- Always

  $\sigma \models \Box p \Leftrightarrow \sigma \models (pUfalse)$

- Eventually

  $\sigma \models \Diamond q \Leftrightarrow \sigma \models (trueUq)$

- Next

  $\sigma[i] \models Xp \Leftrightarrow \sigma_i + 1 \models P$

- Implication

  $p \rightarrow q \models (!p) \vee q$

- Equivalence

  $p \leftrightarrow q \models (p \leftrightarrow q) \wedge (q \leftrightarrow p)$

第4章　構造付き検証パターンの提案

4.1　従来の代表的な検証パターン

この節では現在までに提唱されている代表的検証パターンについての説明を行う．現在の代表的な検証パターンに Dwyer[4] が提唱している検証パターンがある []．この検証パターンはよく使われる時相論理式の記述を性質ごとに分類しパターン化している．分類を図に示す.

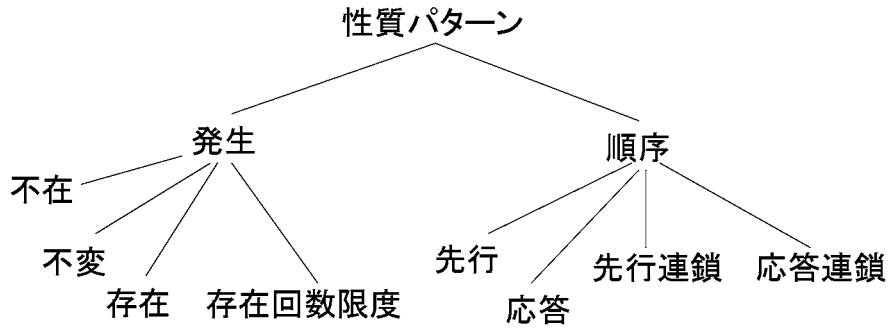


図 4.1: Dwyer が提唱している検証パターン分類

この検証パターンでは SPIN で利用する LTL 式以外の CTL 等の時相論理式に対しても，パターン化しているが，ここでは LTL 式のみを示す.

- 不在...P はずっと偽である.

  [](!P)

- 不変...P はずっと真である.

  [](P)

- 存在...P はいつか真になる.

  <>(P)

- 存在回数限度...P は高々n 回真になる．(以下の例は n=2 時の式)
  (!P W (P W (!P W (P W []!P))))
- 先行...P が真になる前に S が真になる．
  !P W S
- 応答...P が真ならば S がいつか真になる．
  [](P -><>S)
- 先行連鎖
  - P が真になる前に，S が真になり，次にいつか T が真になる．
    <>P ->(!P U (S & !P & X(!P U T)))
  - S が真になり，次にいつか T が真になる前に，P が真になる．
    (<>(S & X<>T)) ->((!S) U P)
- 応答連鎖
  - P が真になり，次にいつか T が真になるならば，P はいつか真になる．
    [](S & X<>T ->X(<>(T & <>P)))
  - P が真にならば，いつか T が真になり，次にいつか P は真になる．
    [](P -><>(S & X<>T))

4.2 本検証パターンの概要

4.1 節で説明した従来の代表的な検証パターンでは LTL のみのパターン化である．本検証パターンでは，UML で特定の構造を持たせ，それに対して性質を定義し，LTL のパターン化を行った．構造と合わせて LTL をパターン化すると性質を具体的にパターン化することができる．また，具体的な UML の構造を LTL と組み合わせることで，ソフトウェア技術者にとって使用しやすくなるという利点もある．詳しくは 4.2 節で説明する．まず，本検証パターンの構成は以下のものである．

- UML から PROMELA への変換規則
- 検証パターンカタログ
  1. 設計モデルに多出する構造を抽象化した構造
  2. 上記の構造でよく検査される性質を LTL で記述

1と2をセットとしてまとめたものを検証パターンカタログとしてまとめた．検証パターンカタログについては4.3節で説明する．図4.2は本検証パターンを使った典型的な検証方法について書いたものである．検証手順は以下のようになる．

1. 設計モデルを作成する．
2. 検証パターンカタログより対応する構造を見つける．
3. 対応した構造でパターン化されている性質で対応する性質を見つける．
4. 設計モデルを検証パターンに対応した変換規則でPROMELAへ変換する．
5. 検証パターンカタログのLTLを設計モデルに依存した形で特殊化し，検証を行う．

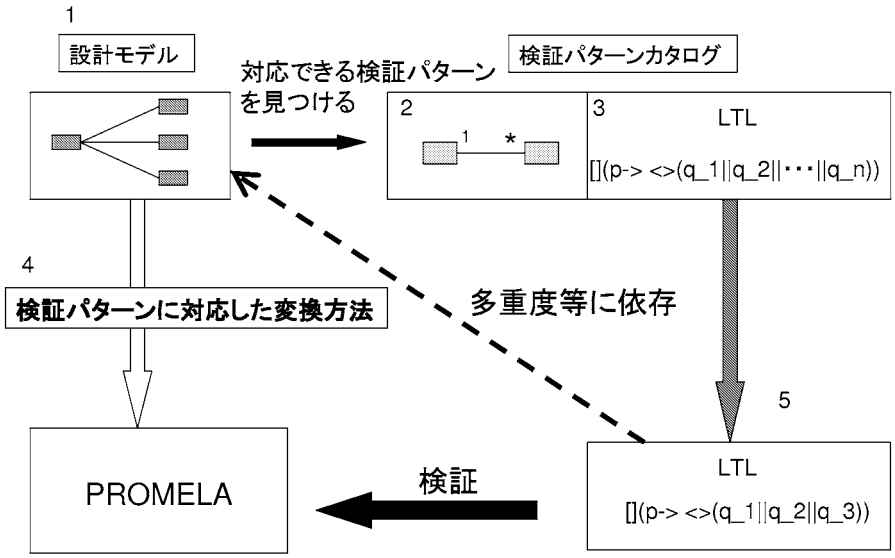


図 4.2: 本検証パターンを使った検証方法

4.3 従来の検証パターンとの比較

従来の検証パターンとの違いは以下の２つある．

- UML上の具体的な性質をパターン化している．
- 対象システムに対してUMLから手続き的にPROMELAへ変換できるようにしている．

それぞれについて説明する．

性質のパターン化
従来の代表的な検証パターンは 4.1 節で説明したように，一般的な性質しかパターン化できない．したがって，汎用性はあるが，ソフトウェア技術者にとっては，ソフトウェア構造と結びつけたパターンのほうが利用しやすい．

仕様記述言語への変換方法の提供　本検証パターンではUMLのクラス図とステートチャート図からSPINの仕様記述言語であるPROMELAに手続き的に変換する方法を与えている．変換方法については次節で説明する．

4.4　UMLからPROMELAへの変換規則

図4.3にクラス単体，図4.4にクラス図，図4.5にステーチャート図，それぞれのPROMELAへの変換を示す．図 4.3 はクラス単体の変換であるが，クラスのメンバ属性を LTL 式で検証できるように，グローバル変数としている．クラスを１つの proctype にし，操作名，定数の返り値は mtype とした．

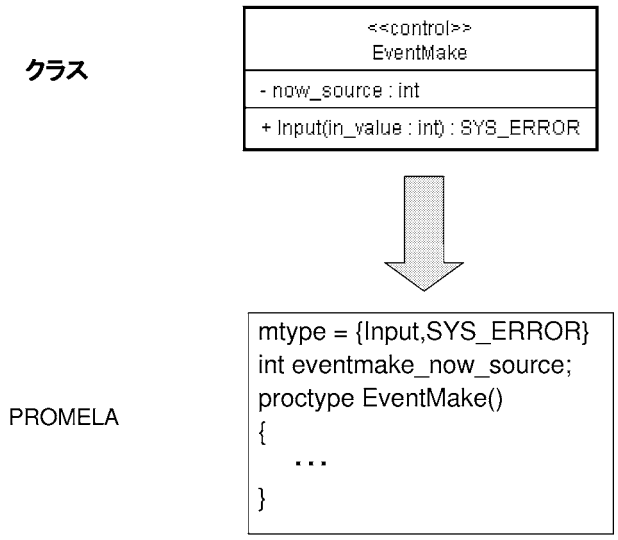


図 4.3: クラスから PROMELA への変換

クラス間の関連に関する変換である．メッソドに関してはチャネルはメッソドの呼び出し用と返り値用の２つをセットして作っている．呼び出しと返り値とはチャネルの型が変わるので，そのための配慮である．

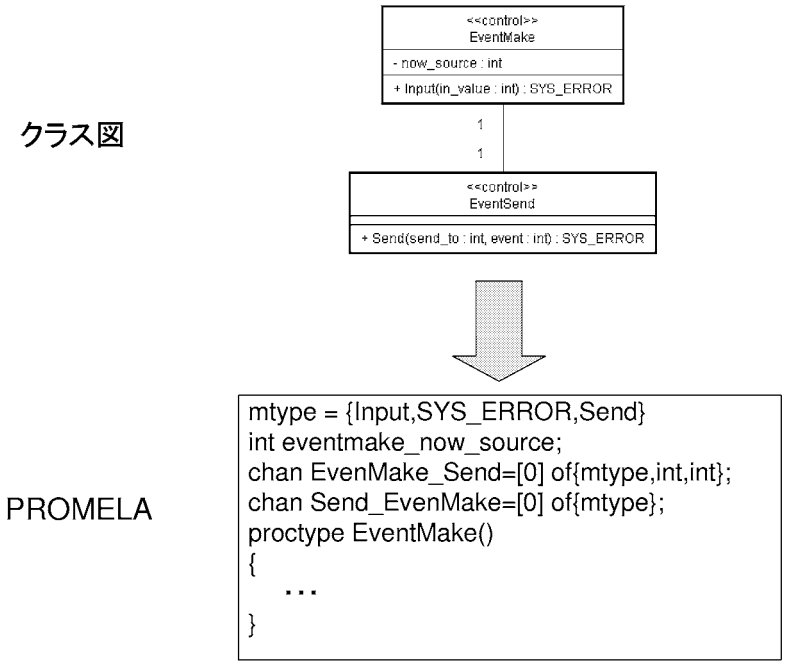


図 4.4: クラス図から PROMELA への変換

ステートチャート図の変換である．各クラスの各状態毎に状態変数を持つ．状態変数は bool 型「クラス名_状態名」としている．この状態変数は自分の状態に入ると true になり，出ると false になる．また，ラベルである「状態名_ENTRY」はその状態に入ったことを示す.「状態名_EXIT」はその状態から出たことを示す．遷移ではどこの状態にもないので，そのクラスの状態変数は全て false になる．

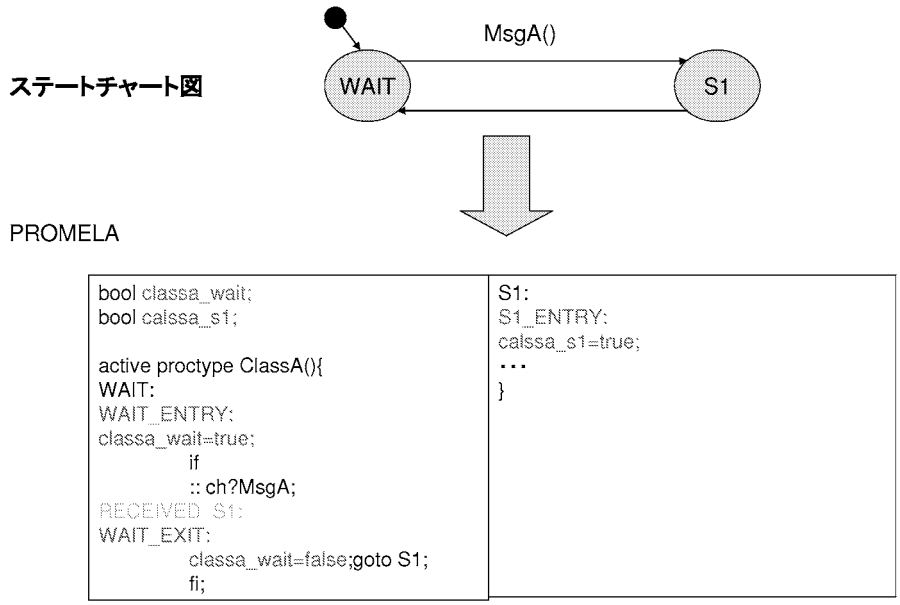


図 4.5: ステートチャート図から PROMELA への変換

性質要素の変換方法

以下に性質要素の PROMELA への変換方法を示す．UML の性質を A，PROMELA の表記を B とした時，A　→　B と記述する．

- 状態... ラベルと状態変数で記述
  - ある状態に入る　→　状態名_ENTRY:　（例　WAIT_ENTRY:）
  - ある状態にいる　→　クラス名_state = true;　（例　class_state = true;）
  - ある状態から出る　→　状態名_EXIT　（例　WAIT_EXIT:）
- 属性... 条件式として記述
  - ある値である　→　クラス名_data==値　（例　Information_data==10）
  - ある値ではない　→　クラス名_data!=値　（例　Information_data!=10）
  - ある値以上 (超える) である　→　クラス名_data>=(>) 値
    （例　Information_data>=(>)10）
  - ある値以下（未満）である　→　クラス名_data<=(<) 値
    （例　Information_data<=(<)10）
- メッセージ... ラベルとして記述

- あるメッセージを送信した　→　SENT_メッセージ名　（例　SENT_NOTICEEVENT:）
- あるメッセージを受信した　→　RECEIVED_メッセージ名
  （例　RECEIVED_NOTICEEVENT:）

4.5　検証パターンカタログ

4.5.1　検証パターンの記述項目

- 名前

  検証パターンの構造が簡潔に連想できる名前を示す。

- 分類

  検証対象になる要素により分類。メッセージ送受信、状態、変数に分類する。

- 構造概要

  この構造は何をするのか、その原理と意図は何か等について。

- 構造

  UML のクラス図で記述。

- 構成要素

  構造に使われているクラスの責任分担。

- 協調関係

  それぞれの構成要素が責任分担を遂行するためにどのように強調するか。ステートチャート図も記述する。

- 検査項目と LTL

  本検証パターンで定義している検証項目とそれを LTL で変換したもの。

- 例題

  簡単な例題を使って検証パターンの使用方法の例を解説する。

- 注意事項

  間違い易い点や意味が理解しにくい点等の補足。

4.5.2 抽象化したLTL式の記述

4.5.3 検証パターンの一覧

今回主に，メッセージの送受信に関する構造と，状態に関する構造に関するに注目して，構造を決定した．属性に注目した構造は本研究では，取り上げなかったので，今後の課題としたい．以下の構造に対して検証パターンをまとめた．

- １－１メッセージ送受信構造
- １－Ｎメッセージ送受信構造
- メッセージ送受信連鎖構造
- クラスの状態管理構造
- リソースの取り合い構造

その中の１つの例を示す．

――――――――――――――――以下は検証パターンカタログから抜粋――――――――――――――――

- 名前

  １－Ｎメッセージ送受信構造

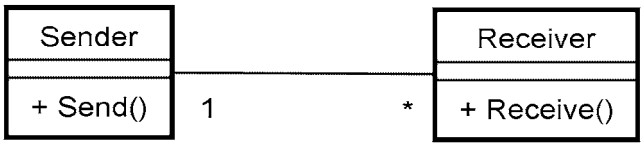
- 分類

  メッセージ送受信

- 構造概要

  １つのメッセージ送信用クラスが複数のクラスにメッセージを送る構造。

- 構造



- 構成要素

- Sender クラス

  受信したメッセージに対して、適したメッセージを、適した複数のクラスに送信する.

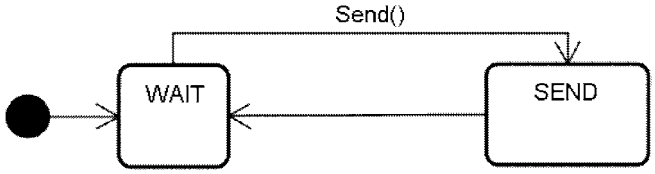
- Receiver クラス

  メッセージを受信する.

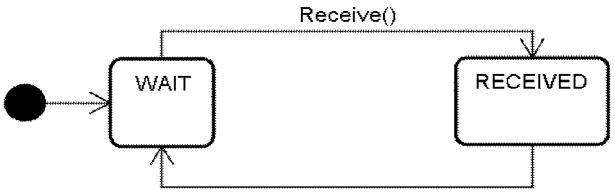
- 協調関係

  Sender クラスが Send メッセージを受信したら、Rceiver クラスに Receive メッセージを送信する。

  - Sender クラス



  - Receiver クラス



- 検査項目と LTL

  - （単純応答）

    送信クラスがメッセージを送信したら、受信クラスの 1 つ以上のインスタンスが受信する。

    LTL

    以下に LTL を示す.

```
#define send Sender@RECEIVED_Send
#define receive (Receiver[pid_1]@RECEIVED_Receive
```

```
                ||Receiver[pid_2]@RECEIVED_Receive
                || ··· ||Receiver[pid_n]@RECEIVED_Receive)

[](send -> <>receive)
```

LTL と要素の対応関係

* send…Sender クラスのインスタンスが Send メッセージを受信した。
* receive…Receiver クラスのインスタンスが receive メッセージを受信した。

– (特定応答)
送信クラスがメッセージを送信したら、受信クラスの特定のインスタンスを含む 1 つ以上が受信する。

LTL
以下に LTL を示す.

```
#define send Sender@RECEIVED_Send
#define receive Receiver[pid_n]@RECEIVED_Receive

[](send -> <>receive)
```

LTL と要素の対応関係

* send…Sender クラスのインスタンスが Send メッセージを受信した。
* receive…Receiver クラスのインスタンスが receive メッセージを受信した。

– （網羅）
送信クラスがメッセージを送信したら、全ての受信クラスのインスタンスが受信する。

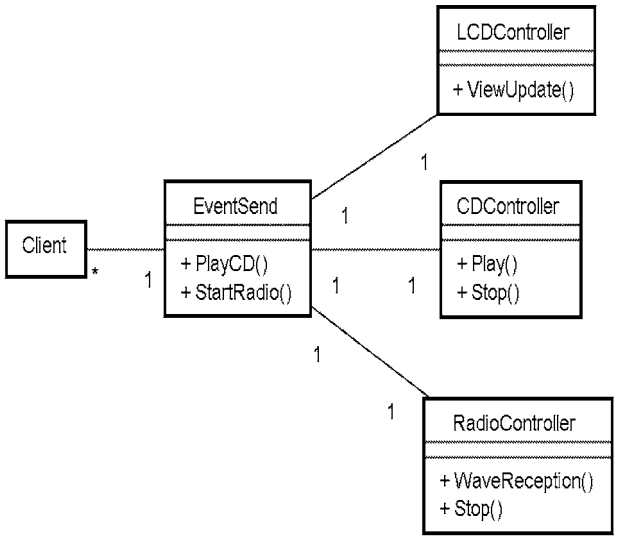
LTL
以下に LTL を示す.

```
#define send Sender@RECEIVED_Send
#define receive_1 Receiver[pid_1]@RECEIVED_Receive
#define receive_2 Receiver[pid_2]@RECEIVED_Receive
```

```
#define receive_n Receiver[pid_n]@RECEIVED_Receive

[](send -> <>receive_1 && <>receive_2 && ··· && <>receive_n)
```

* send…Sender クラスのインスタンスが Send メッセージを受信した。
* receive…Receiver クラスのインスタンスが receive メッセージを受信した。

- 例題例として以下のある CD ステレオの一部の構造を考える。



LTL

– （単純応答）

EventSend クラスが PlayCD メッセージを受信したら、RadioController クラスが Stop メッセージを受信するか、CDController クラスが Play メッセージを受信するか、LCDController クラスが ViewUpdate メッセージを受信する。

```
#define send EventSend@RECEIVED_PlayCD
#define receive (RadioController@RECEIVED_Stop
            ||CDController@RECEIVED_Play
            ||LCDController@RECEIVED_ViewUpdate)

[](p -> <>q)
```

- (特定応答)

  EventSend クラスが PlayCD メッセージを受信したら、少なくとも CDController クラスが Play メッセージを受信する.

  LTL

  以下に LTL を示す.

```
#define send EventSend@RECEIVED_PlayCD
#define receive CDController@RECEIVED_Play

[](send -> <>receive)
```

- （網羅）

  EventSend クラスが StartRadio メッセージを受信を行ったら、LCDController クラスが View メッセージを，CDController クラスが Stop メッセージを，RadioController クラスが WaveReception メッセージをそれぞれ受信する.

  LTL

  以下に LTL を示す.

```
#define send EvedSend@RECEIVED_StartRadio
#define receive_1 LCDController@RECEIVED_View
#define receive_2 CDController@RECEIVED_Stop
#define receive_3 RadioiController@RECEIVED_WaveReception

[](send -> <>receive_1 && <>receive_2 && <>receive_3)
```

- 注意事項
  - メッセージの受信前述の「UML から PROMELA へのマッピング方法」でも記述しているようにメッセージの受信は以下のように記述する。

```
...
channel?MSG ->
RECEIVED_MSG: operational sentence;
...
```

この記述は厳密には、メッセージの受信により真になるのではなく、”operational sentence;”が実行されて真になる。本検証パターンでは、メッセージが受信される順番とメッセージが受信され最初の命令文が実行されることは等価であると考え、上記のような表記をしている。純粋に受信のみを検査したい場合は適当ではない。

以下のように記述すると、メッセージを受信する前にラベルが真になってしまうので、適当ではない。

```
...
RECEIVED_MSG: channel?MSG ->
operational sentence;
...
```

第5章　評価

5.1　評価方法

本研究の評価は企業から提供されたカーオーディオ・システムの設計モデルを使って評価する．具体的には，このカーオーディオ・システムの重要な構造に対して重要な性質を選定し，それを本検証パターンを適用し検証する．

5.2　評価対象

外からユーザーによって入力される情報に対してどの制御にどのメッセージを送ればいいかを，制御している構造に対して検証をおこなった．この検証用に抽象化した設計モデルを図 5.1 に示す．

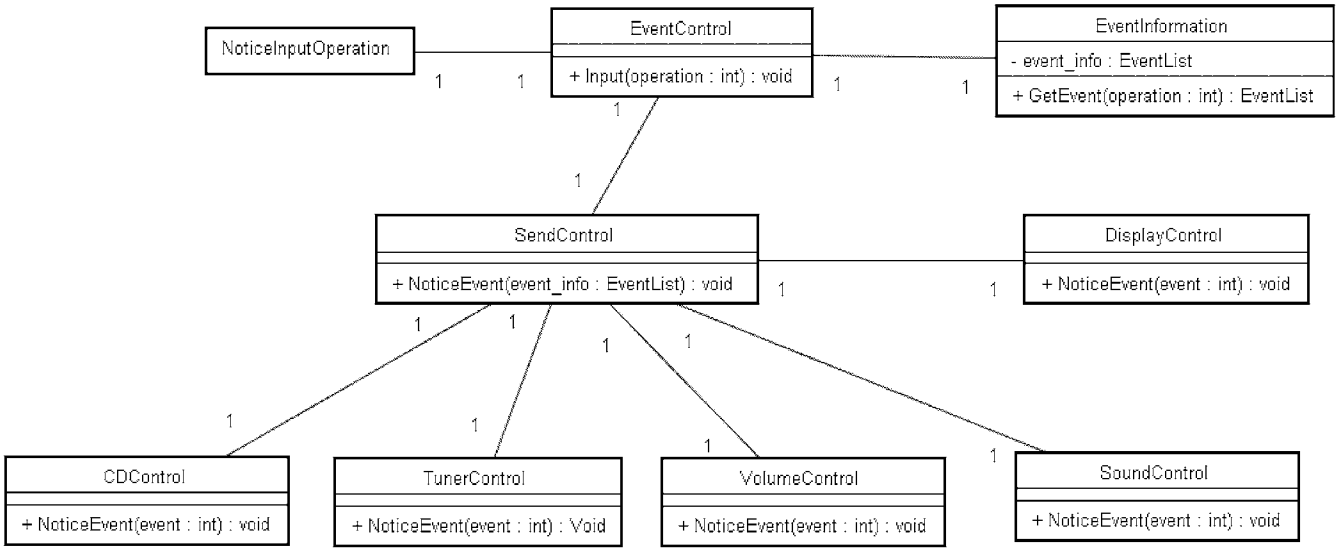


図 5.1: イベント生成構造

構成要素

- NoticeInputOperation クラス

  ユーザーからの入力を受け取り EventControl クラスにその情報を送信する．

- EventControl クラス

  NoticeInputOperation クラスから入力情報を受け取る．その入力情報を EventInformation クラスに送信して，送信先，イベントの情報を受信し，その情報を SendControl クラスに送信する．

- EventInformation クラス

  EventControl クラスから受信した，入力情報に対して，送信先，イベントの情報を EventControl クラスに送信する．

- SendControl クラス

  EventControl クラスから受信した，送信先，イベントの情報に基づいて各送信先に，イベントを送信する．

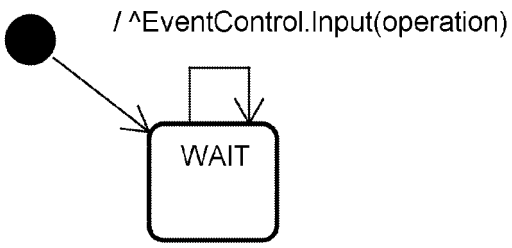
- DisplayControl クラス，CDControl クラス，TunerControl クラス，VolumeControl クラス，SoundControl クラス

  SendControl クラスから各イベントを受信する．

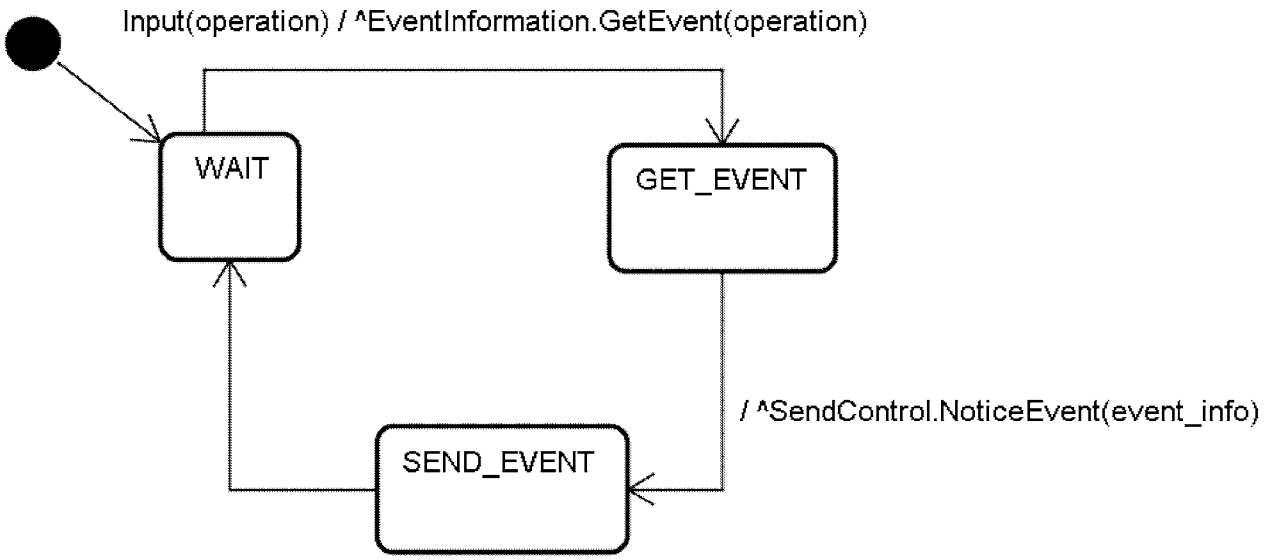
協調関係

EventControl クラスが Input メッセージを受信したら，EventInformaiton クラスに GetEvent メッセージを受信し，属性 event_info を EventControl クラスに返す．EventControl クラスは SendControl クラスに NoticeEvent メッセージを送信する．SendControl クラスは受信した，情報に基づいて，各制御クラスに NoticeEvent メッセージとパラメータを送信する．

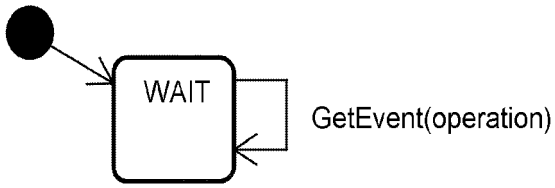
- NoticeInputOperation クラス



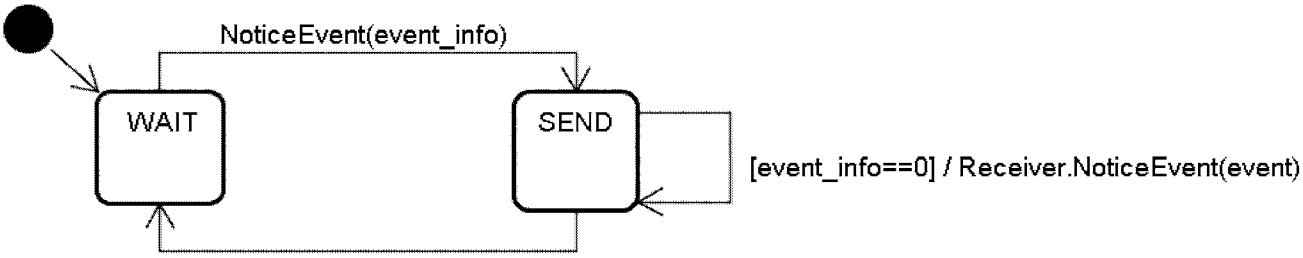
- EventControl クラス



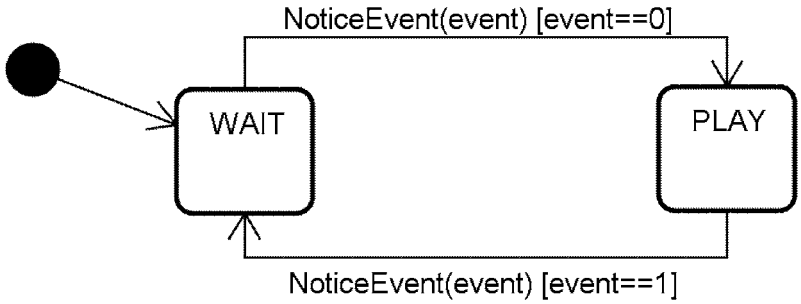
- EventInformation クラス



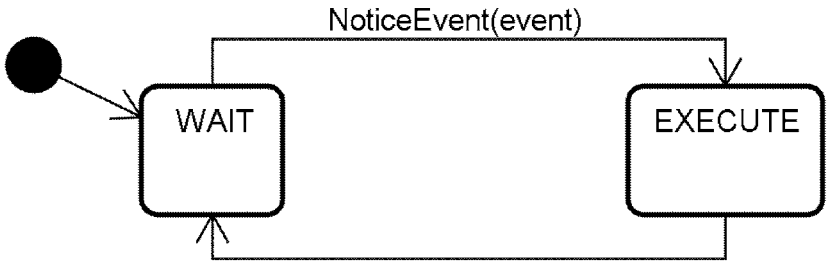
- SendControl クラス



- CDControl クラス,TunerControl クラス



- VolumeControl クラス,SoundControl クラス,DisplayControl クラス



5.2.1 イベント生成構造におけるメッセージの網羅性

SendControl クラスがメッセージ送る各制御クラス全てに，メッセージが遅れているか検証する．検証項目は以下のようになる．

検証項目

SendControl クラスが NoticeEvent メッセージを受信したら，DisplayControl クラス，CDControl クラス，TunerControl クラス，VolumeControl クラス，SoundControl クラス，全てがいつか NoticeEvent メッセージを受信する．

適用する検証パターン

検証パターンカタログから「1-N メッセージ送受信構造」を適用する．検証項目として，「網羅」を適用する．検証パターン「1-N メッセージ送受信構造」の構造を図 5.2 に示す．

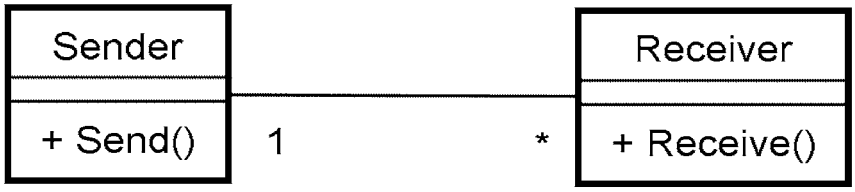


図 5.2: 1-N メッセージ送受信構造

設計モデルと検証パターンのクラスの対応を以下に示す.「設計モデルクラス　→　検証パターンクラス」のように表記する.

- SendControl クラス　→　Sender クラス
- Receiver クラス　→　CDControl クラス, TunerControl クラス, VolumeControl クラス,SoundControl クラス,DisplayControl クラス

検証パターンの LTL

まず, 検証パターンの LTL を以下に示す.

```
#define send EvedSend@RECEIVED_StartRadio
#define receive_1 LCDController@RECEIVED_View
#define receive_2 CDController@RECEIVED_Stop
#define receive_3 RadioiController@RECEIVED_WaveReception

[](send -> <>receive_1 && <>receive_2 && <>receive_3)
```

設計モデルを検証する LTL

次に, 上記の検証パターンの LTL を利用して, 記述した事例の設計モデルを検証する LTL を以下に示す.

```
#define send EvedSend@RECEIVED_StartRadio
#define receive_1 CDControl@RECEIVED_NoticeEvent
#define receive_2 TunerControl@RECEIVED_NoticeEvent
#define receive_3 VolumeControl@RECEIVED_NoticeEvent
#define receive_4 SoundControl@RECEIVED_NoticeEvent
#define receive_5 DisplayControl@RECEIVED_NoticeEvent
```

```
[](send -> <>receive_1 && <>receive_2 && <>receive_3
          && <>receive_4 && <>receive_5)
```

5.2.2 イベント生成構造における実行ソースの排他性

CDContorol クラスと TunerControl クラスが同時に，実行状態になることがないことを検証する．検証項目は以下のようになる．

検証項目

CDControl クラス，TunerControl クラスが同時に状態 PLAY になることはない．

適用する検証パターン

検証パターンカタログから「クラスの状態管理構造」を適用する．検証項目として,「排他」を適用する．検証パターン「クラスの状態管理構造」の構造を図 5.3 に示す．

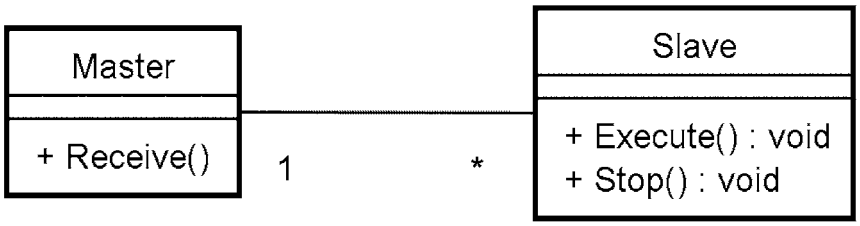


図 5.3: クラスの状態管理構造

設計モデルと検証パターンのクラスの対応を以下に示す.「設計モデルクラス　→　検証パターンクラス」のように表記する．

- SendControl クラス　→　Master クラス
- Slave クラス　→　CDControl クラス，TunerControl クラス

検証パターンの LTL

まず，検証パターンの LTL を以下に示す．

```
#define execute_state_1 slave_1_execute
#define execute_state_2 slave_2_execute
#define execute_state_3 slave_3_execute
```

```
 ...
#define execute_state_n slave_n_execute


[]((!execute_state_1 && !execute_state_2
    && !execute_state_3 && ! && !execute_state_n)
    || (execute_state_1 && !execute_state_2
        && !execute_state_3 && ! && !execute_state_n)
    || (!execute_state_1 && execute_state_2
        && !execute_state_3 && ! && !execute_state_n)
    ||  || (!execute_state_1 && !execute_state_2
              && !execute_state_3 && ! && execute_state_n))
```

設計モデルを検証する LTL

次に，上記の検証パターンの LTL を利用して，記述した事例の設計モデルを検証する LTL を以下に示す.

```
#define execute_state_1 cdcontrol_play
#define execute_state_2 tunercontrol_play


[]((!execute_state_1 && !execute_state_2)
   || (execute_state_1 && !execute_state_2)
   || (!execute_state_1 && execute_state_2))
```

5.3 評価の考察

以上のように実際に本検証パターンを利用して，検証行った．事例を実際に行い，今回の事例に対して，本検証パターンを使った場合と使わなかった場合の違いを定性的にまとめ，下記の表に示す.

以下のように定性的に考察をまとめた．以下は　の場合である.

記述項目	本検証パターンあり	本検証パターンなし
LTL 式の作成	性質の要素の変換方法 (P16)「メッセージ受信」を使って，命題を PROMELA へ変換する．	「SendControl クラスが NoticeEvent メッセージを受信」，「DisplayControl クラスが NoticeEvent メッセージを受信」,「CDControl クラスが NoticeEvent メッセージを受信」,「TunerControl クラスが NoticeEvent メッセージを受信」,「VolumeControl クラスが NoticeEvent メッセージを受信」,「SoundControl クラスが NoticeEvent メッセージを受信」，の PROMELA の表記方法をそれぞれ考える．
	検証パターン「1-N メッセージ送受信構造」の LTL「網羅」を利用する．命題部分に「命題の記述」で変換した PROMELA 表記を命題部分に当てはめる．	検証項目を満たす LTL を時相論理演算子を組み合わせて考える．また，「命題の記述」で考えた PROMELA 表記を命題部分に当てはめる．

本検証パターンに用意してある，性質の要素の変換方法と対象モデルの変換方法を用いることで，パターンなしに比べて，検証者が毎回どう記述するか考える項目が軽減されていることがわかった．

また，本検証パターンでは，注目する性質の要素を状態と属性とメッセージの送受信に絞っている．この３つの性質以外は本検証パターンは対応できない．いくら PLOMELA 変換部分を利用しても，LTL との関連が無く，LTL のパターンが利用できない．この場合，パターンを利用しない場合と同じ労力が必要であると考えている．

第6章　まとめ

6.1　研究のまとめ

本研究では，モデル検査ツール SPIN を用いて，UML を検証する時の対象システムの PROMELA への変換と対象システムに満たしてほしい性質を記述する LTL の記述を支援することを目的とした．解決策として，対象システムの PROMELA への変換については UML のクラス図とステートチャート図から PROMELA へ手続き的に変換する方法を提案した．満たしてほしい性質を記述する LTL の記述についてはソフトウェアの設計でよく用いられる構造に対して，よく満たしてほしい性質を考え，具体的な性質をパターン化しカタログ的にまとめる方法を提案し，実現した．

上記の提案した手法を企業から提供いただいた事例に適用し，評価を行った．今回の評価で行った検証項目に対しては，本手法を適用し検証することができた．

本研究では，２つの成果が得たれた．１つ目は，本研究の目的である，モデル検査を行う際の記述の支援である．本検証パターンを利用することによって，PROMELA の記述，LTL の記述を軽減させることができた．2つ目は，新たな検証パターンの提案である．本検証パターンカタログでまとめた，UML の構造と LTL を合わせて検証パターンとする方法は今回作成した４つの構造以外にも用いることができ，パターン化が行える．

6.2　今後の課題と展望

6.2.1　課題

- カタログの整備

  もちろん，今回カタログかした構造，性質がよく設計等用いられる全てではない．５つの構造についてしか，パターン化できなかったのは，少なかったと考えている．今後，今回のパターン化手法を用いて，もっと多くの構造，性質を検証パターンカタログとすることができると考えている．

6.2.2　展望

- 検証指向的な開発方法論

本研究で提案した検証パターンは現時点では，設計モデルを作成してから，該当する検証パターンを選んで検証する，という手順になっている．しかし，ある性質を満たすようにしたいという要求がまずあった場合，その性質が満たされているか検証しやすい構造を選んで，その構造で設計を行う，という開発方法も可能ではないだろうか．この開発方法論を本検証パターンを基礎として提案していきたいと考えている．

第7章　終わりに

7.1　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] *http://spinroot.com/spin*

[2] *G.J,Holzmann.The SPIN Model Checker.Addison-Wsley 2004*

[3] *E.Clarke,O.Grumberg,and D.Peled : Model Checking, MIT 1999*

[4] *Patterns in Property Specifications for Finite-state Verification, Matthew B. Dwyer, George S. Avrunin and James C. Corbett to appear in Proceedings of the 21st International Conference on Software Engineering, May, 1999.*

付録

検証パターンを以下に示す.

①1-N メッセージ送受信構造

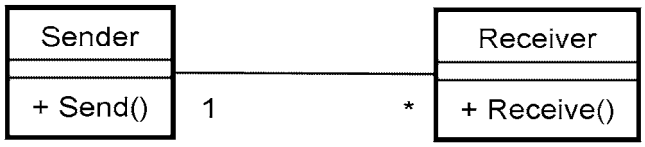
- 分類

  メッセージ送受信

- 構造概要

  １つのメッセージ送信用クラスが複数のクラスにメッセージを送る構造。

- 構造



- 構成要素

  – Sender クラス

  受信したメッセージに対して、適したメッセージを、適した複数のクラスに送信する.

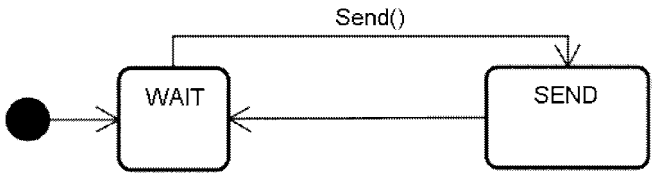
  – Receiver クラス

  メッセージを受信する.

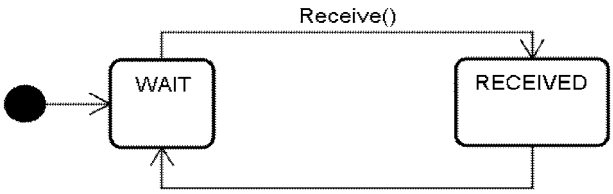
- 協調関係

  Sender クラスが Send メッセージを受信したら、Rceiver クラスに Receive メッセージを送信する。

  – Sender クラス



– Receiver クラス



- 検査項目と LTL
  - （単純応答）
    送信クラスがメッセージを送信したら、受信クラスの 1 つ以上のインスタンスが受信する。

    LTL
    以下に LTL を示す.

```
#define send Sender@RECEIVED_Send
#define receive (Receiver[pid_1]@RECEIVED_Receive
                ||Receiver[pid_2]@RECEIVED_Receive
                || … ||Receiver[pid_n]@RECEIVED_Receive)

[](send -> <>receive)
```

    LTL と要素の対応関係

    * send…Sender クラスのインスタンスが Send メッセージを受信した。
    * receive…Receiver クラスのインスタンスが receive メッセージを受信した。
  - (特定応答)
    送信クラスがメッセージを送信したら、受信クラスの特定のインスタンスを含む 1 つ以上が受信する。

    LTL
    以下に LTL を示す.

```
#define send Sender@RECEIVED_Send
#define receive Receiver[pid_n]@RECEIVED_Receive
```

```
[](send -> <>receive)
```

LTL と要素の対応関係

- send…Sender クラスのインスタンスが Send メッセージを受信した。
- receive…Receiver クラスのインスタンスが receive メッセージを受信した。

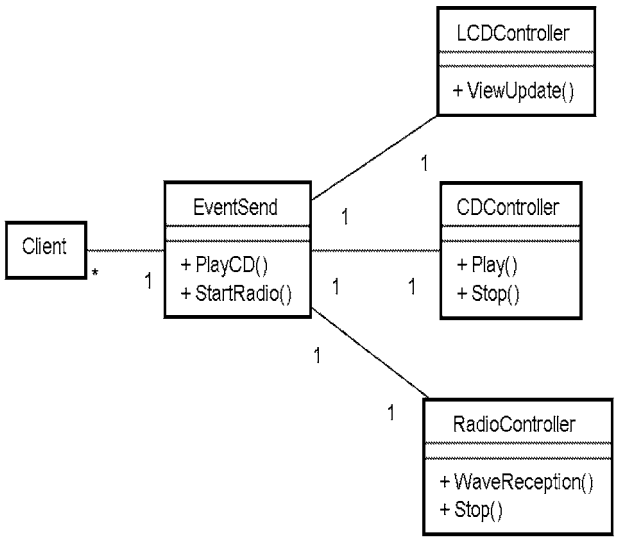
– （網羅）
送信クラスがメッセージを送信したら、全ての受信クラスのインスタンスが受信する。

LTL
以下に LTL を示す.

```
#define send Sender@RECEIVED_Send
#define receive_1 Receiver[pid_1]@RECEIVED_Receive
#define receive_2 Receiver[pid_2]@RECEIVED_Receive

#define receive_n Receiver[pid_n]@RECEIVED_Receive

[](send -> <>receive_1 && <>receive_2 && … && <>receive_n)
```

- send…Sender クラスのインスタンスが Send メッセージを受信した。
- receive…Receiver クラスのインスタンスが receive メッセージを受信した。

- 例題例として以下のある CD ステレオの一部の構造を考える。



LTL

– （単純応答）

EventSend クラスが PlayCD メッセージを受信したら、RadioController クラスが Stop メッセージを受信するか、CDController クラスが Play メッセージを受信するか、LCDController クラスが ViewUpdate メッセージを受信する。

```
#define send EventSend@RECEIVED_PlayCD
#define receive (RadioController@RECEIVED_Stop
            ||CDController@RECEIVED_Play
            ||LCDController@RECEIVED_ViewUpdate)

[](p -> <>q)
```

- (特定応答)

EventSend クラスが PlayCD メッセージを受信したら、少なくとも CDController クラスが Play メッセージを受信する.

LTL

以下に LTL を示す.

```
#define send EventSend@RECEIVED_PlayCD
```

```
#define receive CDController@RECEIVED_Play

[](send -> <>receive)
```

- （網羅）

  EventSend クラスが StartRadio メッセージを受信を行ったら、LCDController クラスが View メッセージを，CDController クラスが Stop メッセージを，RadioController クラスが WaveReception メッセージをそれぞれ受信する．

  LTL

  以下に LTL を示す．

```
#define send EvedSend@RECEIVED_StartRadio
#define receive_1 LCDController@RECEIVED_View
#define receive_2 CDController@RECEIVED_Stop
#define receive_3 RadioiController@RECEIVED_WaveReception

[](send -> <>receive_1 && <>receive_2 && <>receive_3)
```

- 注意事項
  - メッセージの受信前述の「UML から PROMELA へのマッピング方法」でも記述しているようにメッセージの受信は以下のように記述する。

```
...
channel?MSG ->
RECEIVED_MSG: operational sentence;
...
```

    この記述は厳密には、メッセージの受信により真になるのではなく、”operational sentence;”が実行されて真になる。本検証パターンでは、メッセージが受信される順番とメッセージが受信され最初の命令文が実行されることは等価であると考え、上記のような表記をしている。純粋に受信のみを検査したい場合は適当ではない。

    以下のように記述すると、メッセージを受信する前にラベルが真になってしまうので、適当ではない。

```
...
RECEIVED_MSG: channel?MSG ->
```

```
operational sentence;
 ...
```

②メッセージ送受信連鎖構造

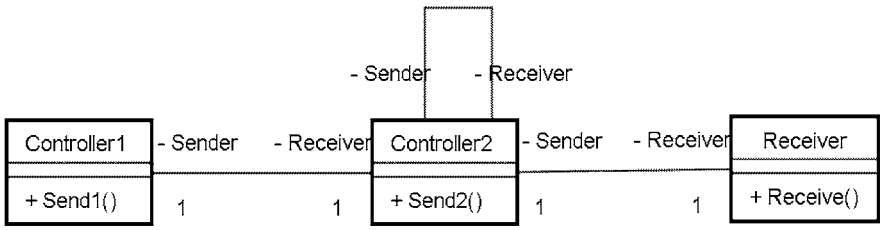
- 分類

  メッセージ送受信

- 構造概要

  送信クラスでもあり受信クラスでもあるクラスがつらなる構造。

- 構造



- 構成要素

  – Controller1 クラス
  受信したメッセージに対して、適したメッセージを、適したクラスに送信する。

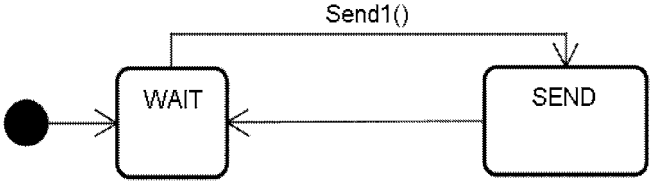
  – Controller2 クラス
  受信したメッセージに対して、適したメッセージを、適したクラスに送信する。

  – Receiver クラス
  メッセージを受信する。

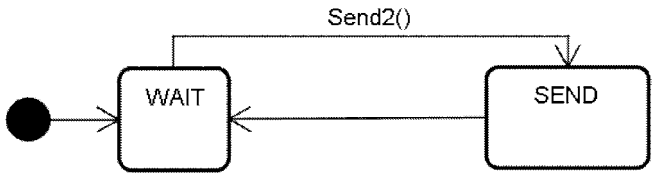
- 協調関係

  Controller1 クラスが Send1 メッセージを受信したら、Controller2 クラスに Send2 メッセージを送信して、Controller2 クラスは ControllerN クラスに SendN メッセージを送信する。このやり取りを繰り返して、最終的に Receiver クラスに Receive メッセージを送信する。

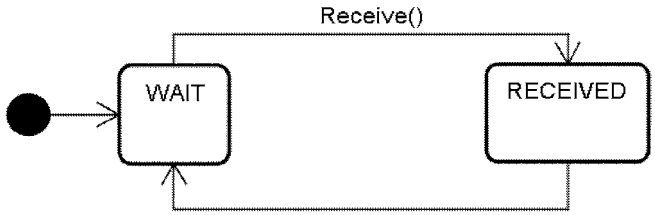
  – Controller1 クラス



– Controller2 クラス



– Controller2 クラス



- 検査項目と LTL
  - （連鎖）

    Controller1 クラスが Send1 メッセージを受信したら、いつか必ず Sender2 クラスが Send2 メッセージを受信し、 Rceiver クラスが Receive メッセージを受信する。

    LTL

    以下に LTL を示す.

```
 #define send_1 Controller_1@RECEIVED_Send_1
#define send_2 Controller_2@RECEIVED_Send_2
#define send_3 Controller_3@RECEIVED_Send_3
 ···
#define send_n Controller_n@RECEIVED_Send_n
#define receive Receiver@RECEIVED_Receive

(<>send_2 && <>send_3 && ··· && <>send_n) ->
[](send_1 -> (((!send_2 && !send_3 && ·          ·          ·
!send_n && !receive) U send1)
```

```
                && (!send_3 && … && !send_n && !receive) U send_2)
                && … && (!send_n U receive) && <>receive)
```

LTL と要素の対応関係

- Send1…Sender クラスのインスタンスが Send1 メッセージを受信した。
- Send2…Sender クラスのインスタンスが Send2 メッセージを受信した。
- receive…Receiver クラスのインスタンスが receive メッセージを受信した

- 例題例として以下のある CD ステレオの一部の構造を考える。

LTL

– （連鎖）

EventSend クラスがメッセージを受信したら、いつか必ず RadioController クラスが Stop メッセージを受信し、CDController クラスが Play メッセージを受信し、LCDController クラスが ViewUpdate() を受信する。

```
 #define send EventSend@RECEIVED_PlayCD
#define first_receive RadioController@RECEIVED_Stop
#define next_receive_1 CDController@RECEIVED_Play
#define last_receive LCDController@RECEIVED_ViewUpdate

(<>first_receive && <>next1_receive) ->
([]send -> ((!next1_receive && !last_receive ) U first_receive)
            && (!last_receive U next1_receive) && <>last_receive)
```

③1-1 メッセージ送受信構造

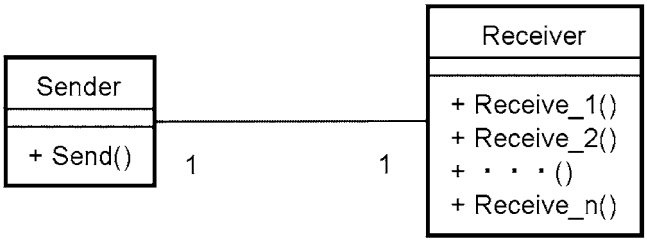
- 分類

  メッセージ送受信

- 構造概要

  １つのメッセージ送信用クラスが１つのクラスに複数のメッセージを送る構造。

- 構造



- 構成要素

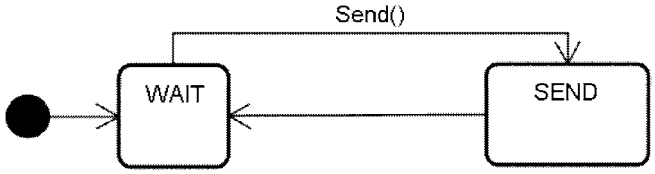
  – Sender クラス
  受信したメッセージに対して、適したメッセージを、適したクラスに送信する.

  – Receiver クラス
  メッセージを受信する.

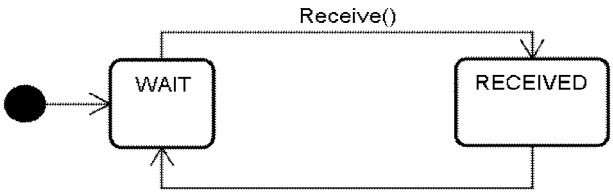
- 協調関係

  Sender クラスが Send メッセージを受信したら、Rceiver クラスに Receive メッセージを送信する。

  – Sender クラス



  – Receiver クラス



- 検査項目と LTL
  - （単純応答）
    送信クラスがメッセージを送信したら、受信クラスが 1 つ以上何かメッセージを受信する。

    LTL
    以下に LTL を示す.

```
#define send Sender@RECEIVED_Send
#define receive (Receiver@RECEIVED_Receive_1
                ||Receiver@RECEIVED_Receive_2
                || ··· ||Receiver@RECEIVED_Receive_n)

[](send -> <>receive)
```

    LTL と要素の対応関係

    * send…Sender クラスのインスタンスが Send メッセージを受信した。
    * receive…Receiver クラスのインスタンスが receive_n メッセージを受信した。
  - (特定応答)
    送信クラスがメッセージを送信したら、受信クラスは特定のメッセージを含む 1 つ以上が受信する。

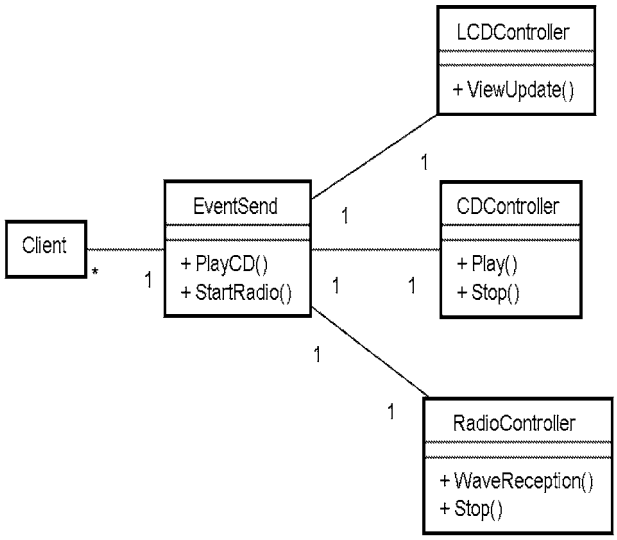
    LTL
    以下に LTL を示す.

```
#define send Sender@RECEIVED\_Send
#define receive Receiver@RECEIVED\_Receive

[](send -> <>receive)
```

LTL と要素の対応関係

* send…Sender クラスのインスタンスが Send メッセージを受信した。
* receive…Receiver クラスのインスタンスが receive メッセージを受信した。

- 例題例として以下のある CD ステレオの一部の構造を考える。



LTL

– （単純応答）

EventSend クラスが PlayCD メッセージを受信したら、RadioController クラスが Stop メッセージを受信するか. または，WaveRception メッセージを受信するか.

```
#define send EventSend@RECEIVED_PlayCD
#define receive (RadioController@RECEIVED_Stop || RadioController@RECEIVE

[](p -> <>q)
```

- (特定応答)

EventSend クラスが PlayCD メッセージを受信したら、少なくとも CDController クラスが Play メッセージを受信する.

LTL

以下に LTL を示す.

```
#define send EventSend@RECEIVED_PlayCD
#define receive CDController@RECEIVED_Play

[](send -> <>receive)
```

④クラスの状態管理構造

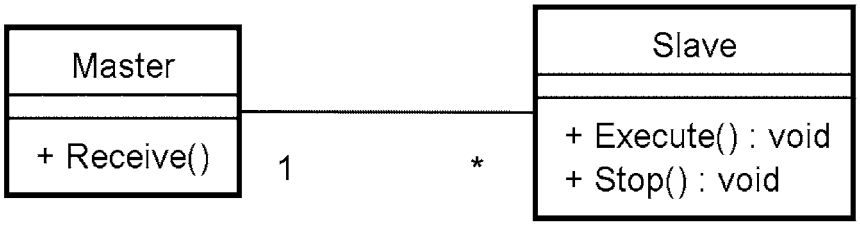
- 分類

  状態

- 構造概要

  １つの管理クラスが複数の被管理クラスにメッセージを送ることによって、状態を管理する。

- 構造



- 構成要素

  - Master クラス

    Slave クラスにメッセージを送って状態を変える。

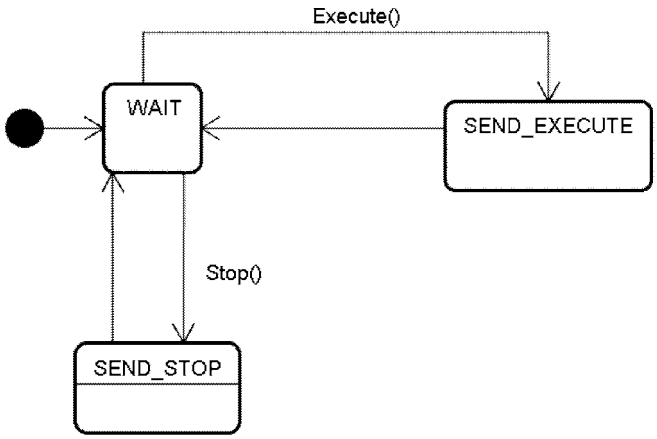
  - Slave ククラス

    Master クラスからメッセージ受信し、自分の状態を変える。

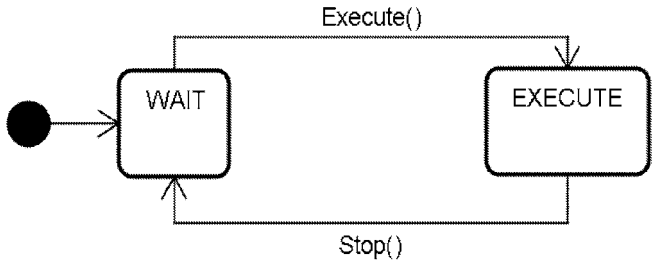
- 協調関係

  Master クラスが Receive メッセージを受信したら、Slave クラスに適したメッセージを送信する。Slave クラスは Execute メッセージを受信すると、状態 Execute に、Stop メッセージを受信すると状態 WAIT にそれぞれなる。

  - Master クラス



– Slave クラス



- 検査項目と LTL

– （排他）

Slave クラスの 1 つのインスタンスのみが、状態 EXECUTE である。

LTL

以下に LTL を示す.

```
 #define execute_state_1 slave_1_execute
#define execute_state_2 slave_2_execute
#define execute_state_3 slave_3_execute
 ...
#define execute_state_n slave_n_execute


[]((!execute_state_1 && !execute_state_2
```

```
    && !execute_state_3 && ! && !execute_state_n)
    || (execute_state_1 && !execute_state_2
        && !execute_state_3 && ! && !execute_state_n)
    || (!execute_state_1 && execute_state_2
        && !execute_state_3 && ! && !execute_state_n)
    ||  || (!execute_state_1 && !execute_state_2
              && !execute_state_3 && ! && execute_state_n))
```

LTL と要素の対応関係

＊ execute_state1…Slave クラスのインスタンスが状態 EXECUTE になった。

＊ execute_state2…Slave クラスのインスタンスが状態 EXECUTE になった。

– （特定排他）

ある Slave クラスのインスタンスが状態 EXECUTE の時、ある Slave クラスのインスタンスは状態 EXECUTE にならない。

LTL

以下に LTL を示す.

```
 #define execute_state1 Slave1_execute
#define execute_state2 Slave2_execute

[]!(execute_state1 && execute_state2)
```

LTL と要素の対応関係

＊ execute_state1…Slave クラスのインスタンスが状態 EXECUTE になった。

＊ execute_state2…Slave クラスのインスタンスが状態 EXECUTE になった。

– （状態応答）

ある Slave クラスのインスタンスが状態 EXECUTE ならば、ある Slave クラスのインスタンスはいつか状態 EXECUTE になる。

LTL

以下に LTL を示す.

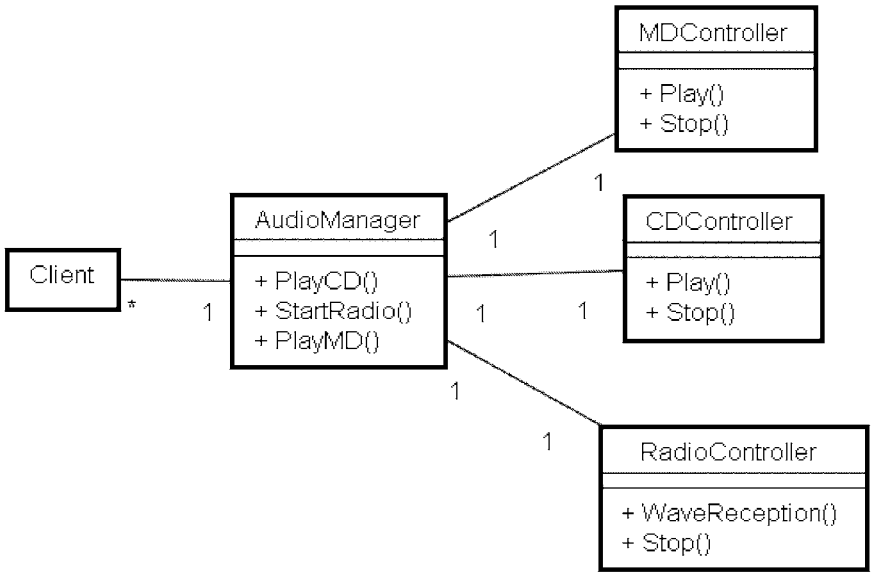
```
#define execute_state1 slave1_execute
```

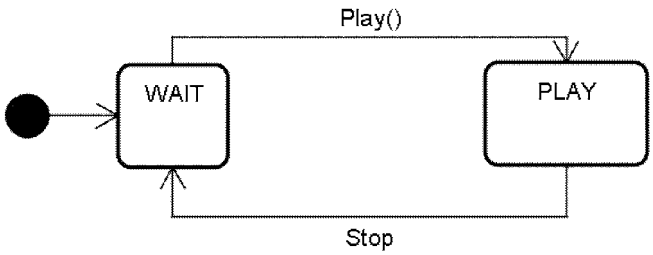
```
#define execute_state2 slave2_execute
#define execute_state3 slave3_execute
 ...

[](execute_state1 -> (<>execute_state2 (&& <>execute_state3)*)
```

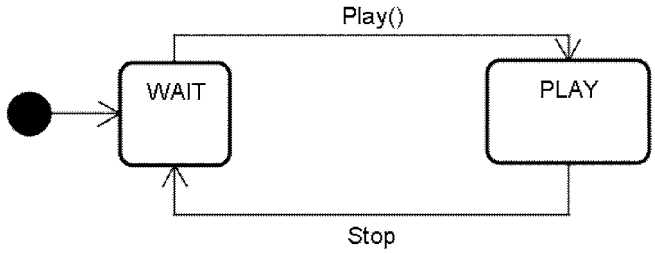
LTL と要素の対応関係

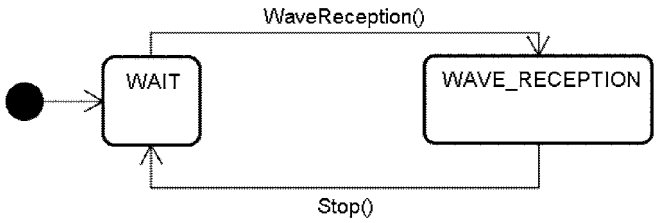
- execute_state1…Slave クラスのインスタンスが状態 EXECUTE になった。
- execute_state2…Slave クラスのインスタンスが状態 EXECUTE になった。

- 例題例として以下のある CD ステレオの一部の構造を考える。









LTL

– （排他）

CDController クラスのみが、状態 Play である。

```
#define execute_state1 mdcontroller_play
#define execute_state2 cdcontroller_play
#define execute_state3 radiocontroller_wavereception

[]!(execute_state1 && execute_state2 && execute_state3)
```

– （特定排他）

CDController クラスが、状態 Play である時、MDController クラスは状態 Play にならない。

```
#define execute_state1 mdcontroller_play
#define execute_state2 cdcontroller_play

[]!(execute_state1 && execute_state2)
```

```
```

– （状態応答）

MDController クラスが状態 Play ならば、CDController クラスはいつか状態 Play になる.

```
#define execute_state1 mdcontroller_play
#define execute_state2 cdcontroller_play

[](execute_state1 -> (<>execute_state2)
```

⑤リソースの取り合い構造

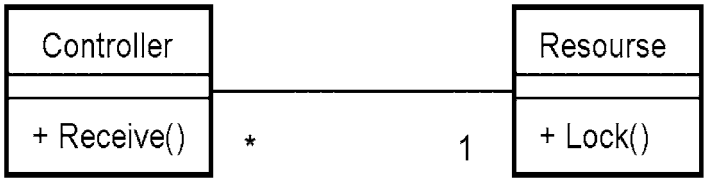
- 分類

  状態

- 構造概要

  複数のクラスがリソースを取り合う構造。

- 構造



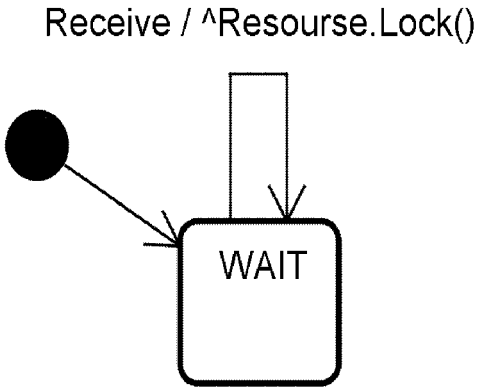
- 構成要素
  - Controller クラス

    Resourse クラスにメッセージを送り、リソースを使用する。
  - Resourse ククラス

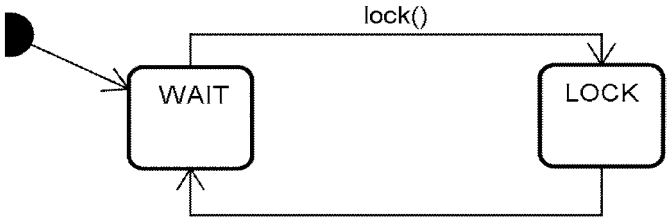
    Resourse クラスにメッセージを送り、リソースを使用する。

- 協調関係

  Controller クラスが Lock メッセージを Resourse クラスに送信すると、状態 Lock になる。処理が終わると状態 WAIT に戻る。
  - Controller クラス



– Resourse クラス



- 検査項目と LTL
  - （応答性）

    Resource クラスが状態 LOCK になったら、必ず状態 WAIT になる。

    LTL

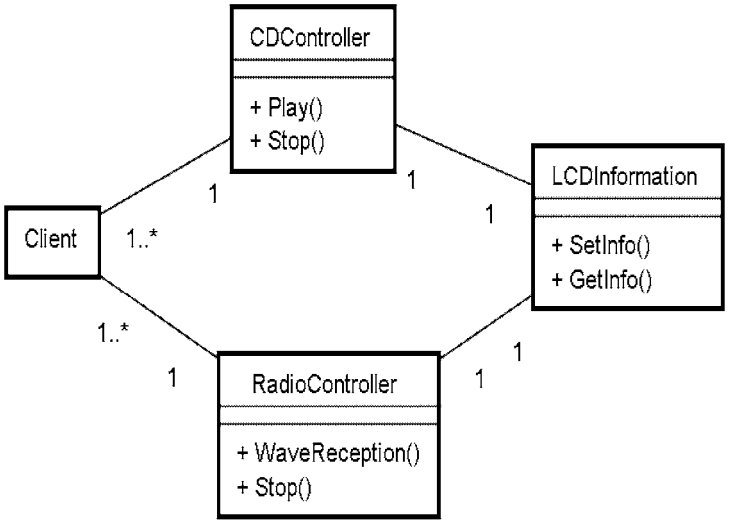
    以下に LTL を示す.

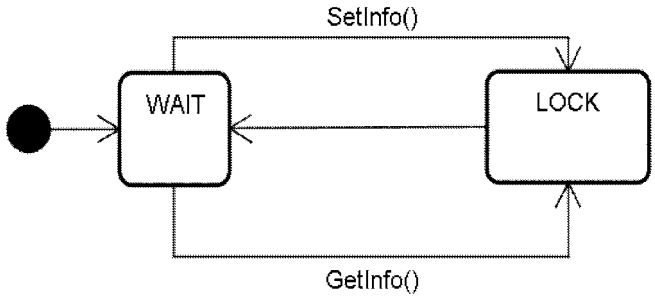
```
  #define lock resourse_lock;
#define wait resourse_wait

[](lock -> <>wait)
```

LTL と要素の対応関係

* lock…Resourse クラスが状態 LOCK になっている。
* wait…Resourse クラスが状態 WAIT（状態 LOCK でない）になっている。

- 例題例として以下のある CD ステレオの一部の構造を考える。





LTL

– （応答性）

LCDInformation クラスが状態 LOCK になったら、必ずいつか状態 WAIT になる。

```
#define lock lcdinformation_lock
#define wait lcdinformation_wait

[](lock -> <>wait)
```

評価で使用したPROMELAコードを以下に示す.

```
#define WAIT 0
#define GETEVENT 1
#define SENDEVENT 2
#define SEND 3
#define PLAY 4
#define EXECUTE 5


mtype={Input,GetEvent,NoticeEvent}

chan NoticeInputOperation_EventControl_Input_call = [10] of {mtype,int}

chan EventControl_EventInformation_GetEvent_call = [0] of {mtype,int}
chan EventControl_EventInformation_GetEvent_return = [0] of {byte}

chan EventControl_SendControl_NoticeEvent_call = [0] of {mtype,byte}
chan EventControl_SendControl_NoticeEvent_return = [0] of {bool}

chan SendControl_CDControl_call = [5] of {mtype,int}

chan SendControl_TunerControl_call = [5] of {mtype,int}

chan SendControl_VolumeControl_call = [5] of {mtype,int}

chan SendControl_SoundControl_call = [5] of {mtype,int}

chan SendControl_DisplayControl_call = [5] of {mtype,int}


bool noticeinputoperation_wait;

bool eventcontrol_wait;
bool eventcontrol_get_event;
bool eventcontrol_send_event;

bool eventinformation_wait;
```

```
bool sendcontrol_wait;
bool sendcontrol_send;

bool cdcontrol_wait;
bool cdcontrol_play;

bool tunercontrol_wait;
bool tunercontrol_play;

bool volumecontrol_wait;
bool volumecontrol_execute;

bool soundcontrol_wait;
bool soundcontrol_execute;

bool displaycontrol_wait;
bool displaycontrol_execute;


proctype NoticeInputOperation(){
int operation;
byte goto_state;

Wait:
Wait_Entry: noticeinputoperation_wait=true;
    if
        :: operation=1;
        :: operation=2;
        :: operation=3;
        :: operation=4;
        :: operation=5;
    fi;
Sent_Input:    NoticeInputOperation_EventControl_Input_call!Input,operation;
    atomic{noticeinputoperation_wait=false;goto_state = WAIT;}

    noticeinputoperation_wait=true;

Wait_Exit:
```

```
noticeinputoperation_wait=false;
    if
    :: goto_state==WAIT -> goto Wait;
    fi;
}

proctype EventControl(){
int operation;
int getevent;

Wait:
Wait_Entry: eventcontrol_wait=true;
    if
        :: NoticeInputOperation_EventControl_Input_call?Input,operation ->
Received_Input:
Sent_GetEvent:    atomic{noticeinputoperation_wait=false;
                EventControl_EventInformation_GetEvent_call!GetEvent,operation;goto

    eventcontrol_wait=true;

Wait_Exit:
    eventcontrol_wait=false;
    if
    :: goto_state==GETEVENT -> goto Get_Event;
    fi;

Get_Event:
Get_Event_Entry: eventcontrol_get_event=true;
    if
        :: NoticeInputOperation_EventControl_Input_return?getevent ->
            atomic{noticeinputoperation_wait=false;
                    EventControl_EventInformation_GetEvent_call!GetEvent,operation;

    eventcontrol_get_event=true;

Wait_Exit:
eventcontrol_get_event=false;
    if
```

```
    :: goto_state==SETEVENT -> goto Set_Event;
    fi;


Set_Event:
Set_Event_Entry: eventcontrol_set_event=true;
if
        :: NoticeInputOperation_EventControl_Input_return?getevent ->
            atomic{noticeinputoperation_wait=false;
            EventControl_EventInformation_GetEvent_call!GetEvent,operation;goto_sta

    eventcontrol_set_event=true;

Wait_Exit:
eventcontrol_set_event=false;
    if
    :: goto_state==WAIT -> goto Wait;
    fi;
}


proctype EventInformation(){
int operation;
byte goto_state;

Wait:
Wait_Entry: EventInformation_wait=true;

    EventControl_EventInformation_GetEvent_call?GetEvent,operation;
Received_GetEvent:
    atomic{noticeinputoperation_wait=false;goto_state = WAIT;}
Sent_GetEvent:
    EventControl_EventInformation_GetEvent_return!operation;
    n_wait=true;

Wait_Exit:
noticeinputoperation_wait=false;
    if
```

```
    :: goto_state==WAIT -> goto Wait;
    fi;
}

proctype SendControl(){
int operation;
byte goto_state;

Wait:
Wait_Entry: EventInformation_wait=true;

    EventControl_EventInformation_GetEvent_call?GetEvent,operation;
Received_GetEvent:
    atomic{noticeinputoperation_wait=false;goto_state = WAIT;}
Sent_GetEvent:

    if
    :: operation == 1 -> EventSend_CDControl_NoticeEvent_call!NoticeEvent,operation
    :: operation == 2 -> EventSend_TunerControl_NoticeEvent_call!NoticeEvent,operat
    :: operation == 3 -> EventSend_VolumeControl_NoticeEvent_call!NoticeEvent,opera
    :: operation == 4 -> EventSend_SoundControl_NoticeEvent_call!NoticeEvent,operat
    :: operation == 5 -> EventSend_DisplayControl_NoticeEvent_call!NoticeEvent,oper

    n_wait=true;

Wait_Exit:
noticeinputoperation_wait=false;
    if
    :: goto_state==WAIT -> goto Wait;
    fi;
}



proctype CDControl(){
int event;

Wait:
```

```
Wait_Entry: cdcontrol_wait=true;
    if
        :: SendControl_CDControl_NoticeEvent_call?NoticeEvent,0 ->
Received_NoticeEvent: goto_state=PLAY

Wait_Exit:
    cdcontrol_wait=false;
    if
    :: goto_state==GETEVENT -> goto Get_Event;
    fi;

Play:
Play_Entry: cdcontrol_play=true;
    if
        :: SendControl_CDControl_NoticeEvent_call?NoticeEvent,1;
Received_NoticeEvent:    goto_state = WAIT;

    eventcontrol_get_event=true;

Play_Exit:
eventcontrol_get_event=false;
    if
    :: goto_state==SETEVENT -> goto Set_Event;
    fi;

}

proctype TunerControl(){

Wait:
Wait_Entry: tunercontrol_wait=true;
    if
        :: SendControl_TunerControl_NoticeEvent_call?NoticeEvent,0 ->
Received_NoticeEvent: goto_state=PLAY

Wait_Exit:
    tunercontrol_wait=false;
    if
```

```
    :: goto_state==GETEVENT -> goto Get_Event;
    fi;

Play:
Play_Entry: cdcontrol_play=true;
    if
        :: SendControl_TunerControl_NoticeEvent_call?NoticeEvent,1;
Received_NoticeEvent:    goto_state = WAIT;

    eventcontrol_get_event=true;

Play_Exit:
eventcontrol_get_event=false;
    if
    :: goto_state==SETEVENT -> goto Set_Event;
    fi;
}


VolumeControl(){
int event;

Wait:
Wait_Entry: Volume_wait=true;
    if
        :: SendControl_VolumeControl_NoticeEvent_call?NoticeEvent,event ->
Received_NoticeEvent: goto_state=EXECUTE

Wait_Exit:
    tunercontrol_wait=false;
    if
    :: goto_state==Execute -> goto Execute;
    fi;

Execute:
Execute_Entry:
    atomic{Volume_execute=true;goto_state = WAIT;}
```

```
Play_Exit:
volumecontrol_execute=false;
    if
    :: goto_state==WAIT -> goto WAIT;
    fi;
}



SoundControl(){
int event;

Wait:
Wait_Entry: SoundControl_wait=true;
    if
        :: SendControl_SoundControlControl_NoticeEvent_call?NoticeEvent,event ->
Received_NoticeEvent: goto_state=EXECUTE;

Wait_Exit:
    tunercontrol_wait=false;
    if
    :: goto_state==Execute -> goto Execute;
    fi;

Execute:
Execute_Entry:
    atomic{SoundControl_execute=true;goto_state = WAIT;}

Play_Exit:
SoundControlcontrol_execute=false;
    if
    :: goto_state==WAIT -> goto WAIT;
    fi;
}


DisplayControl(){
int event;
```

```
Wait:
Wait_Entry: DisplayControl_wait=true;
    if
        :: SendControl_DisplayControlControl_NoticeEvent_call?NoticeEvent,event ->
Received_NoticeEvent: goto_state=EXECUTE

Wait_Exit:
    tunercontrol_wait=false;
    if
    :: goto_state==Execute -> goto Execute;
    fi;

Execute:
Execute_Entry:
    atomic{DisplayControl_execute=true;goto_state = WAIT;}

Play_Exit:
DisplayControlcontrol_execute=false;
    if
    :: goto_state==WAIT -> goto WAIT;
    fi;
}
```