入退出管理システム：ガードコマンドと黒板によるモデルの考察

佐原　伸

タオベアーズ

目次

1 ガ

入退出
モデルと
基本的
ストケー
doOneC
ひとつ
スケジ
ことを促
各ガー
件が true
する。
スケジ
に設定し
各ガー

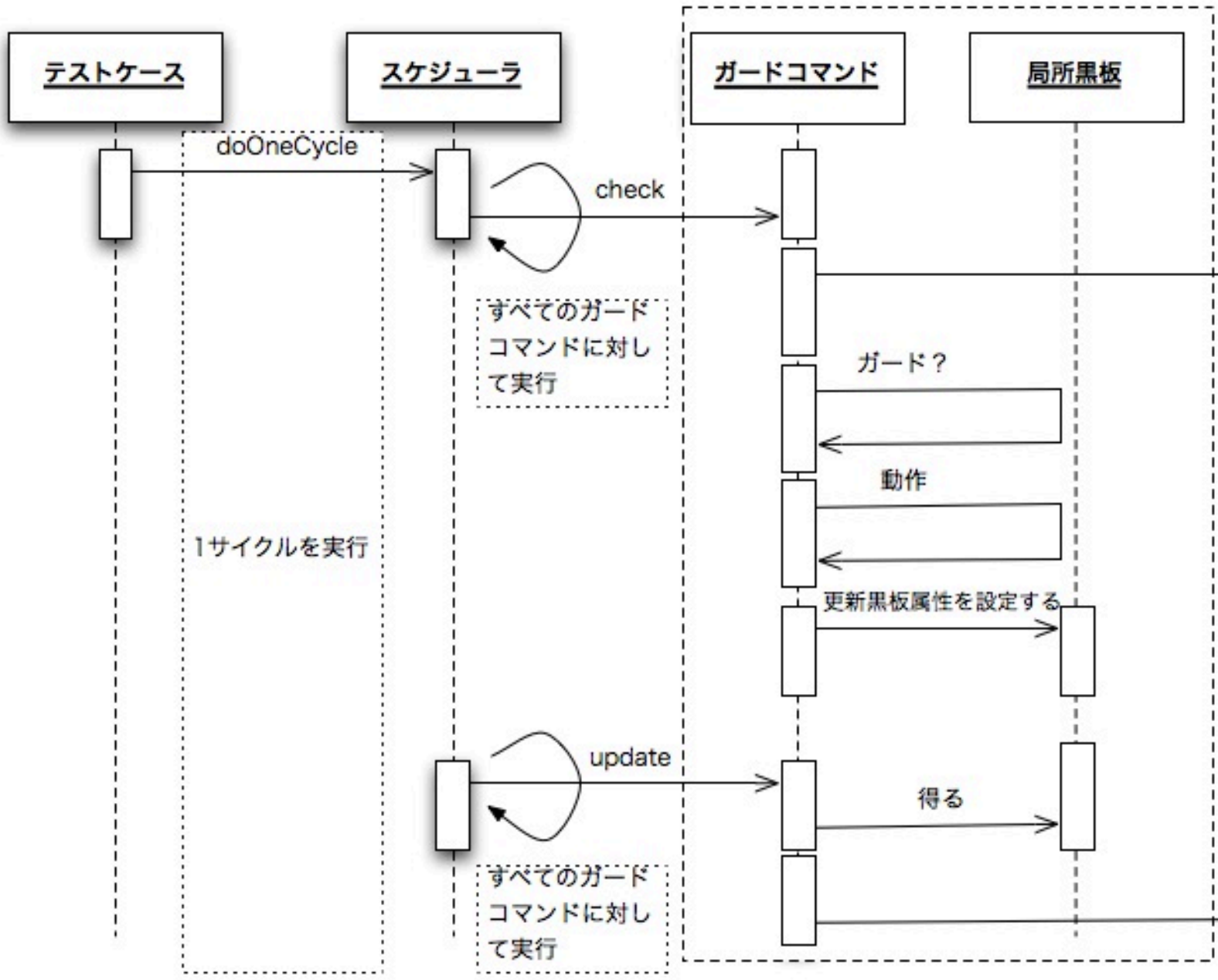

図 1　本フレームワークのシーケンス図

今回のモデル化では、仕様の変更、再利用、保守などを容易に行えるように、仕様を図 2 に示す階層に従って記述した。
以下に、下位階層から概要を説明する。

*1 DB は、利用者がどこにいるかなどの情報を写像で持っている。
*2 黒板属性 DB に直接設定すると、各ガードコマンド動作結果により副作用が起こり得るため、更新された属性だけを持つ。

図 2　モデル記述の階層

1.1　黒板フレームワーク

黒板フレームワークは、黒板クラス[*3]一つからなる。黒板は、並行処理仕様を擬似的に記述するため、ガードコマンドから参照および更新される属性を持つ。

黒板には、システム全体で一つしかない大域黒板と、各ガードコマンド毎に持つ局所黒板がある。

黒板は、本来の属性を持つ黒板属性 DB と、更新時の副作用を避けるために持っている更新黒板属性 DB を持つ。黒板属性 DB は、すべてのガードコマンドの処理が終わるまで更新されない黒板属性 DB である。更新黒板属性 DB は、各ガードコマンドが更新する黒板属性 DB である。

さらに、各黒板属性は、実際の利用者居場所 DB, 仮想世界利用者居場所 DB、ドア DB、案内表示 DB、ID 端末 DB、ID カード DB、センサー DB、タッチパネル DB を持つ。

黒板は、上記の属性を得たり、設定する操作を持っている。また、黒板は「実際と仮想の利用者居場所が等しい」かどうか判定したり、「データの重複がない」かを判定するといった、要求性質の検証や仕様の検証用の操作を持つ。

1.2　ガードコマンド・フレームワーク

ガードコマンド・フレームワークは、ガードコマンドの種類だけクラスが存在し、各ガードコマンド・クラスのインスタンスも通常複数存在する。

各各ガードコマンドは、check と update の 2 つの操作を持ち、

check 操作は、以下の部分からできている。

- 大域黒板から黒板属性 DB を得る
- ガード条件を判定する
- 対応する動作を行う
- 局所黒板の更新黒板属性 DB に設定する

update 操作は、以下の部分からできている。

- 局所黒板から更新黒板属性 DB を得る
- 大域黒板の黒板属性 DB に設定する

1.3　案内サービス

案内アルゴリズムを記述する。

今回は、ガードコマンドとして記述した。

本来は、ID 認証やセンサーから得られる情報だけで構築した、仮想世界の利用者居場所の情報から案内すべきであるが、後述の 2.2 節で述べるとおり、正しい位置情報を得ることが非常に困難であるため、適切に案内ができないので、実際の利用者居場所

[*3] 以下、クラスとそのインスタンス・オブジェクトの区別は、特に必要ない限り記述しない。

の情報から案内を作成している。

1.4 利用者行動

ガードコマンド内部の動作[*4]として記述した。

1.5 要求性質検査

回帰テストケースで、黒板クラスで定義した「実際と仮想の利用者居場所が等しい」関数と、「実際と仮想の利用者居場所を比較表示する」操作で要求性質を呼び出すことにより検査する。

[*4] VDM++[?] の関数あるいは操作呼び出しと代入文で記述した。

2　ガードコマンドと黒板によるモデル化の感想

2.1　本フレームワークの利点

本フレームワークを使用した結果、各ガードコマンドによるガード条件の比較と、対応する動作の記述は、比較的単純で理解しやすく、検証しやすい。

また、当初もくろみ通り、仕様の変更を行った場合の変更余波が小さいことを実感した。

逐次型で並行処理モデルを記述できるため、同様の問題について筆者が過去に行った、他の手法 (2.3 参照) に比べ検証が簡単である。

さらに、VDM++ を使用することにより、事前条件と事後条件のチェックが可能になり、モデルの問題点を早期に発見できた (3.2 参照)。

2.2　実際の利用者居場所と仮想世界の利用者居場所について

センサーや ID 認証によって、仮想世界の利用者居場所情報を得ることを試みたが、場所内に利用者が複数いる場合、センサー情報だけでは、誰がどこにいるかを確定することは非常に難しく、実際に役立つのは ID 認証の情報だけであり、実際の利用者居場所と仮想世界の利用者居場所の乖離がすぐに大きくなってしまった。

そのため、現在のモデルでは、案内アルゴリズムは、実際の利用者居場所から作成している。これを、仮想世界の利用者居場所情報を元に行うと、ほぼ意味のない動きをしてしまうことが分かった。

2.3　他の方法との比較

筆者が過去に行った「状態遷移マシンが並行に動作するシステム」のモデル化と今回の方式の比較を述べる。

2.3.1　VICE[?] を使用した ACSTAM モデルとの比較

VDM++ を非同期並行処理が記述できるように拡張した VICE を使用し、（有）デザイナーズデンの酒匂寛氏が作成した ACSTAM と呼ぶフレームワークを使ってモデル化を行ったが、事前・事後条件による検証 (3 節参照) は、使用できるものの、実行テストでは、テキスト・ファイルに出力されるログをレビューしての検証しか行えず、パラメータを少し変更するだけで、ログの内容が大きく変わり、ある程度大きなモデルになると、検証が非常に困難になった。

仕様記述自体は、フレームワークが用意している状態遷移マシンのサブクラスとして、例題を模倣して記述すればよいので本フレームワークと同様記述しやすい。

2.3.2　プライオリティ・キューとイベントによる割込処理によるモデルとの比較

このモデルは、まだある程度試行しただけで、完全なモデル化は行っていないが、一つの状態遷移マシンを実行するプロセス（あるいはスレッド）のプライオリティキューとイベンによるト割込処理を組み合わせ、逐次処理で「状態遷移マシンが並行に動作するシステム」を記述しようとするもので、Z による同様のモデル化 [1] も行われている。

この形のモデル化を、強い型付けの静的な言語である VDM++ で行う場合、プロセス内で処理を実行中に割込が発生し、他のプロセスに制御が移ることは、素直にモデル化できない。

そこで、

1. プライオリティ・キューとイベントによる割込処理によるプロセスの制御を記述するモデルと、
2. 各プロセス内の処理を記述するモデル

を別々に作らなければならず、両モデルを、レビュー以外で統一的に検証することができない。

1. の制御方式を記述し、検証するには便利だが、アプリケーション仕様ともいうべき 2. の記述・検証には向いていないと言える。

3　validation 方法について

本モデルの検証と validation は下記の方法で行った。

- 実行テスト
- 事前・事後条件
- 証明課題

3.1　実行テスト

VDM++ の標準の検証手段である実行テストで、検証と validation を行った。テストケースは VDM++ で記述し、オランダ・チェス社の Marcel Verhoef 氏が作成した回帰テスト・ライブラリを使用して、あらかじめ設定した検証結果と実行テストの結果が一致するかを見た。

また、実行途中の重要情報は、標準出力にログとして表示し、意図したとおりの実行がなされているかを確認した。

VDM++ 仕様のコードカバレージは 90% 以上を達成した。

3.1.1　デバッグの工夫

特に利用者クラスには「s 前にいた場所」「s 作成者名」という 2 つのインスタンス変数を持たせ、ある利用者オブジェクトをどのオブジェクトが作成し、前にいた場所がどこかを確認しやすいようにした。これにより、仕様のミスをかなり検出できるようになった。

3.2　事前・事後条件

主要な操作と関数には事前条件あるいは事後条件を記述し、テスト実行時にこれら条件を検査することで、仕様のミスを早期に発見できるようにした。

例えば、下記の VDM++ ソースは、黒板に属性を設定する操作であるが、ここで、事前・事後条件の双方で、オブジェクト参照による問題（4.1 参照）から、DB に「ID の重複がある」ことを検出し、どのような場合に起こるかの検出を容易にした。

```
public 設定する　: オブジェクト型 * DB ==> ()
設定する(a オブジェクト型, a 追加 DB) == (
  cases a オブジェクト型 :
    (mk_token("実際の利用者居場所")), (mk_token("仮想世界利用者居場所")) ->
      s 黒板属性 DB(a オブジェクト型) := 利用者居場所 DB から削除する (s 黒板属性 DB(a オブジェクト型), a 追加 DB),
    (mk_token("ドア")) ->
      s 黒板属性 DB(a オブジェクト型) := ドア DB から削除する (s 黒板属性 DB(a オブジェクト型), a 追加 DB),
    (mk_token("案 内 表 示")), (mk_token("ID　端 末")), (mk_token("ID　カ ー ド ")), (mk_token("タ ッ チ パ ネル")), (mk_token("センサー")) ->
      s 黒板属性 DB(a オブジェクト型) := DB から削除する (s 黒板属性 DB(a オブジェクト型), a 追加 DB),
    others ->
      s 黒板属性 DB(a オブジェクト型) := (dom a 追加 DB) <-: s 黒板属性 DB(a オブジェクト型)
  end;
  s 黒板属性 DB(a オブジェクト型) := s 黒板属性 DB(a オブジェクト型) ++ a 追加 DB;
  skip
)
pre
  let wDB = s 黒板属性 DB(a オブジェクト型) in
  a オブジェクト型 in set dom s 黒板属性 DB and
  cases a オブジェクト型 :
  (mk_token("実際の利用者居場所")), (mk_token("仮想世界利用者居場所")) ->
    利用者 ID の重複がない (wDB),
  others ->
```

```
    ID の重複がない（wDB）
  end
post
  let wDB = s 黒板属性 DB(a オブジェクト型) in
  cases a オブジェクト型 :
  (mk_token("実際の利用者居場所")), (mk_token("仮想世界利用者居場所")) ->
    利用者 ID の重複がない（wDB),
  others ->
    ID の重複がない（wDB）
  end;
```

3.3 証明課題

証明課題を生成しレビューすることで、下記 3.3.1 節以下で説明するように、仕様の誤りをいくつか検出した。
これらの証明課題以外にも、いくつかのタイプの証明課題から、事前・事後条件を導出し、仕様中に記述した。

3.3.1 値が空でないことや 0 でないこと

束縛された値が空でないといった証明課題から、不注意により、空集合にはならないはずの箇所で、空集合になるバグをいくつか発見できた。

3.3.1.1 空である場合のチェック

下記の例では、証明課題をレビューすることで、利用者集合が空であるケースを見落としていた仕様のミスを検出した。

```
(forall a 利用者集合 : set of 利用者 &
a 利用者集合 <> {} =>
 (exists w 次の利用者 in set a 利用者集合 &
(forall w 利用者 in set a 利用者集合 &
w 次の利用者.s 到着時間 <= w 利用者.s 到着時間)))
```

修正後の VDM++ ソースは以下の通りである。

```
static public 到着時間が一番早い利用者を選ぶ : set of 利用者 -> 利用者
到着時間が一番早い利用者を選ぶ（a 利用者集合） ==
  let w 次の利用者 in set a 利用者集合 be st
    forall w 利用者 in set a 利用者集合 & w 次の利用者.s 到着時間 <= w 利用者.s 到着時間
  in
  w 次の利用者
pre
  a 利用者集合 <> {}
post
  forall w 利用者 in set a 利用者集合 & RESULT.s 到着時間 <= w 利用者.s 到着時間;
```

3.3.1.2 分母が 0 である場合のチェック

下記の証明課題をレビューすることで、分母が 0 になってしまうケースを見落としていたバグを発見できた[*5]。

```
(not (s 躊躇する最大時間 = 0) => s 躊躇する最大時間 <> 0)
```

[*5] このケースは、VDM の別の処理系 Overture Tool（http://www.overturetool.org/twiki/bin/view）で発見した。現在の VDMTools は過去にはこの類の証明課題を生成していたが、今は。何故か表示しない。

修正後の VDM++ ソースは以下の通りである。

```
public 移動する気になった時間である : () ==> bool
移動する気になった時間である () ==
  def w 乱数 = MATH`rand(s 躊躇する最大時間);
    w 時間の乱れ =
      if s 躊躇する最大時間 = 0 then
        0
      else
        w 乱数 / s 躊躇する最大時間;
    w 移動する気になった時間 = s 躊躇する最大時間 * w 時間の乱れ
  in (
  def - =
    debug >= 9 => new IO().echo(
      "\n「移動する気になった時間である」で生成された乱数=" ^
      VDMUtil`val2seq_of_char[int](w 乱数)^
      "\n"
    )
  in skip;
  return s 現在時間 >=  w 移動する気になった時間 + s 到着時間;
)
```

3.3.2 subtype が適切であること

演算の中間結果が、不注意によって int でなく real になっていたことに気がつき、影響がないかどうか、該当箇所を調べているうちに、VDM の処理系によって結果が異なる箇所を発見することができ、ツールの欠陥を発見できた。

具体的には、3.3.1.2 節と同じ下記のソースで、「w 時間の乱れ」の型が real になっている影響を調査している内に、rand 関数の返値が VDMTools と Overture Tool で異なることを発見し、Overture Tool のバグを検出した。

```
public 移動する気になった時間である : () ==> bool
移動する気になった時間である () ==
  def w 乱数 = MATH`rand(s 躊躇する最大時間);
    w 時間の乱れ =
      if s 躊躇する最大時間 = 0 then
        0
      else
        w 乱数 / s 躊躇する最大時間;
    w 移動する気になった時間 = s 躊躇する最大時間 * w 時間の乱れ
  in (
  以下、省略...
```

3.3.3 map application

写像の domein 側の値が不適切であるケースを発見した。

3.3.3.1 写像の適用

下記の証明課題をレビューすることで、該当の操作に、証明課題と同じ事前条件が必要なことに気づいた。

```
a 居場所 ID in set dom s 移動可能場所集合

public 移動可能な次の場所 ID を得る : 利用者のいる場所 DB * 場所 ID ==> [場所 ID]
移動可能な次の場所 ID を得る (a 利用者のいる場所 DB, a 居場所 ID) == (
  for all w 次の場所 ID 候補 in set s 移動可能場所集合 (a 居場所 ID) do
```

```
    if 利用者が移動可能である（a 利用者のいる場所 DB, w 次の場所 ID 候補） then
      return w 次の場所 ID 候補
    else
      skip;
  return nil
)
pre
  a 居場所 ID in set dom s 移動可能場所集合;
```

4　VDM コーディング上のコツ

4.1　オブジェクト参照の問題

本モデルでは、黒板中に持つ属性を写像で保持し、そのドメインあるいは値域にオブジェクトを使用している場合が多い。

この場合、黒板属性中のオブジェクト内容を変更して更新黒板属性としても、オブジェクト参照のため元の黒板属性の内容も書き換わってしまう。

このため、新たなオブジェクトを作成して、黒板属性中のオブジェクト内容をコピーすると、写像中では異なるオブジェクトとなるため、削除してから新規に追加する形で更新しなければならない。

ところが、旧黒板の内容を参照して削除しようとすると、２つのガードコマンドが同じ黒板属性を同じタイミングで更新する場合、update 時に最初のガードコマンドが削除・更新したオブジェクトを、２つめのガードコマンドは削除前のオブジェクトを参照して削除しようとするため、削除できずに同じ ID(例えば利用者 ID) を持つオブジェクトを２つ作ってしまう。

そこで、本モデルでは、各オブジェクトの ID [*6] を設定し、写像中のオブジェクトを更新する場合、check 操作中で、下記 4.1.1 節のように、写像中のオブジェクトをコピーして、新たに同じ値を持ったオブジェクトを作成して属性を更新し、update 操作から呼び出す「設定する」操作で、下記 4.1.2 節のように、写像中のオブジェクトを削除 (4.2 節を参照) してから、新しいオブジェクトを追加するようにした。

4.1.1　オブジェクトのコピー

```
public check : 黒板 ==> ()
check（a 大域黒板） == (
    ...
    dcl w 更新 ID 端末 : ID 端末 := wID 端末. コピーする（w 位置）;
    dcl w 更新開くべきドア : ドア := w 開くべきドア. コピーする（mk_(f, t)）;
    w 更新開くべきドア. 解錠する ();
    w 更新 ID 端末. 利用可能にする ();
    s 局所黒板. 更新黒板属性を設定する（mk_token("ドア"), {mk_(f, t) |-> w 更新開くべきドア}）;
    s 局所黒板. 更新黒板属性を設定する（mk_token("ID 端末"), {w 更新 ID 端末. 位置を得る () |-> w 更新 ID 端末}）;
    ...
```

4.1.2　オブジェクトの削除と追加

```
public 設定する　: オブジェクト型 * DB ==> ()
設定する（a オブジェクト型，a 追加 DB） == (
    cases a オブジェクト型 :
    (mk_token("実際の利用者居場所")), (mk_token("仮想世界利用者居場所")) ->
        s 黒板属性 DB(a オブジェクト型） := 利用者居場所 DB から削除する（s 黒板属性 DB(a オブジェクト型），a 追加 DB），
    (mk_token("ドア")) ->
        s 黒板属性 DB(a オブジェクト型） := ドア DB から削除する（s 黒板属性 DB(a オブジェクト型），a 追加 DB），
    (mk_token("案内表示")), (mk_token("ID 端末")), (mk_token("ID カード ")),
    (mk_token("タッチパネル")), (mk_token("センサー")) ->
        s 黒板属性 DB(a オブジェクト型） := DB から削除する（s 黒板属性 DB(a オブジェクト型），a 追加 DB），
    others ->
        s 黒板属性 DB(a オブジェクト型） := (dom a 追加 DB） <-: s 黒板属性 DB(a オブジェクト型）
    end;
    s 黒板属性 DB(a オブジェクト型） := s 黒板属性 DB(a オブジェクト型） ++ a 追加 DB;
)
```

この方式の欠点は、各クラスに「コピーする」操作を作成しなければならない点と、各オブジェクトによって、写像からの具体的削除方法が異なるため、仕様の行数がやや長くなる点であるが、本モデルでは、できるだけ共通化するようにし、実際の利用者

[*6] 言語処理系が自動付与するオブジェクト ID でなく、仕様記述者が設定する ID

居場所 DB とドア DB およびその他の DB の 3 パターンに集約できた。

4.2　写像からの削除方法

写像中のオブジェクトの削除を定義域削減 (<−:) 演算子で行うと、同じオブジェクト ID(VDM++ 処理系が付与する ID) を持つオブジェクトを等しいと判断するため、新たに作成した同じ ID(例えば利用者 ID) を持つオブジェクトを異なるオブジェクトと判断し、同じ ID のオブジェクトが写像中に 2 つ作成されてしまう。

そこで、4.2.1 節のように、オブジェクトの ID 集合と削除すべき ID の集合から、削除後の ID 集合を求め、次に、その ID を持つ集合を求め、さらに、値域限定演算子を使って、削除後の写像を求める。

4.2.1　DB からの削除

```
public DB から削除する : 場所関連 DB * 場所関連 DB -> 場所関連 DB
DB から削除する(aDB, a 削除 DB) ==
    if aDB <> {|->} then
        let w 集合 = rng aDB,
            wID 集合 = {w.ID を得る() | w in set w 集合},
            w 削除集合 = rng a 削除 DB,
            w 削除 ID 集合 = {w.ID を得る() | w in set w 削除集合},
            w 削除後 ID 集合 = wID 集合 \ w 削除 ID 集合,
            w 削除後集合 = {w | w in set w 集合 & w.ID を得る() in set w 削除後 ID 集合}
        in
        aDB :> w 削除後集合
    else
        aDB;
```

4.3　今回のモデル実装上の問題点

黒板属性を写像とオブジェクトで表現したため、上記 4.1 節と 4.2 節の問題が発生した。

そこで、黒板の属性を、写像とオブジェクトでなく、集合とレコードで表すことが考えられる。この方式は、今回は採用できなかったが、他のアプリケーションでは成功したことがあり、今後研究する価値があると考える。

5　参考文献等

参考文献

[1] Lain D.Craig. *Formal Method of Operating System Kernel.* Springer, 2007.