CSC227
Formal Specification of Software(7)

John Fitzgerald
Centre for Software Reliability
University of New Castle

Translated by
Takahiko Ogino
Railway Technical Research Institute

Based upon the book
Modelling Systems: Practical Tools and Techniques in Software Development
(ISBN 0 521 623480 Cambridge University Press 1998)
by John Fitzgerald and Peter Gorm Larsen (IFAD)

凡例

- 本翻訳は、John Fitzgerald博士（Centre for Software Reliability、University of New Castle）の講義用OHPを翻訳したものである。
- 翻訳にあたっては、なるべく忠実な訳をしたつもりである。ただし原稿が、授業の教材であることを勘案し、出来るだけ意味的な注釈を付けたり、空白になっておりクラスで埋められるのであろうと思われる部分は、出来るだけ講義のベースとなっている本、Modelling Systemsからの引用によって中身を埋めた。
- そのような部分をオリジナルと区別するため、フォントの色を変化させることで出来るだけ区別している。
- また、同じ鉄道総研の寺田夏樹君には、日本語と内容のプルーフリーディングをお願いしたことを記載し感謝の意を表します。
- 原OHPの修正も含めて全ての日本語版内容に関する責任は、荻野にあります。
- 誤訳も含めて、内容に関するご指摘をお待ちしています。

形式的モデルを確認（Validate)する

- Validation(確認、確証、批准)についての考え
- 内部整合性
  - タイプをチェックするレベル
  - 証明責務
- アニメーション
- システマチックな試験
- 証明
- 確認技術の選択

確認についての考え方

- 形式的モデルが顧客が望んだシステムについて正確に記述しているかに関して、どの程度信頼できるか?
  - 要求仕様は多くの場合不完全で曖昧である:モデル作成者は、曖昧性の無いモデルによって、これらを解決しなければならない。
  - 要求仕様では顧客の意図がしばしば不正確に述べられている。

  →現実問題として仕様を確定するときに一番重要な問題。形だけ仕様確定しても将来的に顧客は文句を言う。

- 確認(Validation)は、モデルが検討中のシステムの正確な表現であるという確信を高めるプロセスを言う。これには、2つの面がある:
  - 1.モデルの内部整合性のチェック→記述されている機能がデータタイプ不変式を遵守し不可能な計算を引き起こさない。
  - 2.モデルがモデル化されているシステムの必要な振る舞いについて正確に記述されているかのチェック→顧客の意図しているように動作するか

確認についての考え方　内部整合性

- もし、モデル化する言語がフォーマルなら、次の規則を持つ:
  - フォーマルなシンタックス:言語におけるシンボルを制約したり、その組み立て方を規定する規則:意味を持たない記号の規則
  - フォーマルなセマンティクス:フォーマルなシンタックスに従って書かれたモデルの意味を決定するための規則:記号列に意味をもたせる
- もし、シンタックスが形式的な場合、ツール(プログラミング言語コンパイラー中のシンタックス・チェッカのように)を用いて、それをチェックすることができる。
- もし、意味論が形式的な場合、ツール(プログラミング言語コンパイラー中のタイプチェッカのように)を用いて、少なくともいくつかのものはチェックすることができる。
- しかしこのような形でのすべてのチェックは不可能である!→そのため、テストデータを入れての実行時のチェックが必須となる。

確認についての考え方　機能の振る舞い

- 確認のもう一つの側面は、モデルが希望のシステムの機能的な振る舞いを正確に記録しているか否かのチェックである。

- 3つのアプローチがある:
- **モデルのアニメーション**– モデル化の概念に慣れていない顧客にはうまくいくが、よいインターフェースが必要である。
- **モデルのテスト**– モデルの適用範囲を評価することができる、しかしテストの質に制限されることと、モデルが実行可能である必要がある。
- **モデルの特性の証明**– 適用範囲の評価においては非常に優れているし、モデルが実行可能でなくても良い。しかしながら、よいツールがあまり無い。

内部整合性　タイプチェック

- 内部整合性をチェックする単純な形は、タイプチェックである。
- 関数定義から抜粋された次のものに有効に作用するタイプチェックのツールを考えなさい:

```
Student :: …

Sid = token

Dbase = map Sid to Student

newStudent1: Sid * Student * Dbase -> Dbase
newStudent1(sid,s,db) == db ++ { sid |-> s }
newStudent2: Sid * Student * Dbase -> Dbase
newStudent2(sid,s,db) == db ^ { sid |-> s }
newStudent3: Sid * Student * Dbase -> Dbase
newStudent3(sid,s,db) == db munion { sid |-> s }
```

内部整合性　タイプチェック

```
Student :: …

Sid = token

Dbase = map Sid to Student

newStudent1: Sid * Student * Dbase -> Dbase
newStudent1(sid,s,db) == db ++ { sid |-> s }
```

- ++オペレータ(オーバーライド)は、マップに対するオペレータであり、かつこのオペレータは、パーシャルではない。タイプチェックOK

```
newStudent2: Sid * Student * Dbase -> Dbase
newStudent2(sid,s,db) == db ^ { sid |-> s }
```

- ^オペレータは、シーケンスをコンカチネーションするオペレータであり、マップタイプには適用できない。

内部整合性　タイプチェック

```
newStudent3: Sid * Student * Dbase -> Dbase
newStudent3(sid,s,db) == db munion { sid |-> s }
pre sid not in set dom db
```

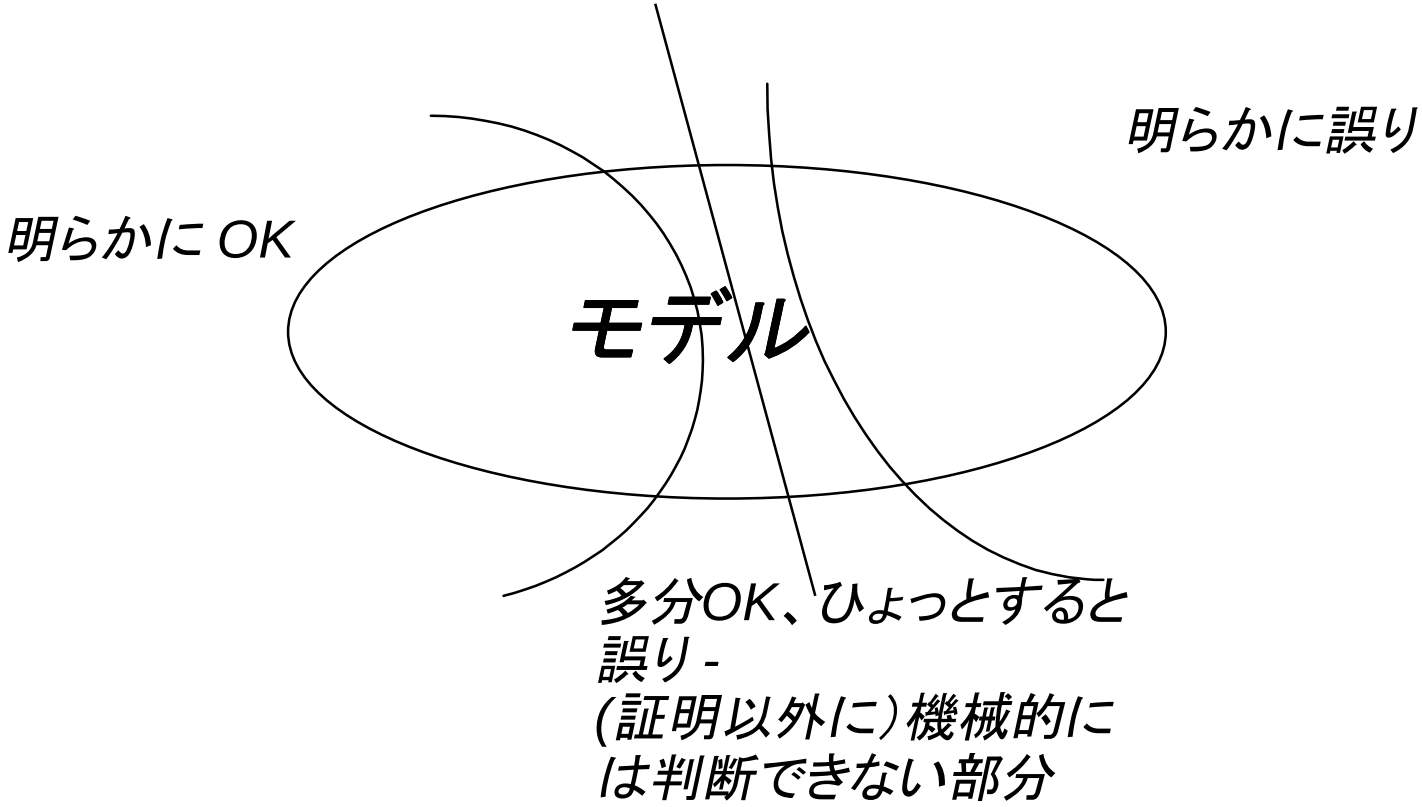
人間は、これで良いことを知っているが、機械も同様に証明出来るだろうか?
　次の場合はどうなるか

```
newStudent3: Sid * Student * Dbase -> Dbase
newStudent3(sid,s,db) == db munion { sid |-> s }
pre sid not in set {s1 | s1 : Sid &
    exists y in set rng db & db(s1) = y}
```

どのようなオペレータの使われ方に対しても、タイプが適切に保護されているかを自動的にチェックできる完全に一般的なツールは無い。

(プログラミング言語も同じ問題を持っている。大変制約され、また表現力が落ちる言語を使用していない限りは、0による割り算が発生するか否かを自動的に静的にチェックする一般的なツールを作ることは出来ない。)。

内部整合性　タイプチェック



- フォーマルなモデリングの現在の研究の多くは、自動的により多くのチェックを行うことによって、真中のエリアのサイズを減らすための技術やツールの開発である。

内部整合性　　証明責務

- チェックを自動的に実行することができない場合、数学的な証明の技術がそれを完成するために用いられる。
- VDMモデル上で実行される全てのチェックの集まりは、**証明責務**と呼ばれる。
- 証明責務は、VDM−SLモデルが、形式的に、内部的に無矛盾であるとみなすために成立することを示す必要がある論理式である。
- 私たちは、VDM−SLモデルで3つの証明責務を見る。
  - 領域のチェック
  - 関数の直接定義におけるSatisfiability
  - 関数の陰定義におけるSatisfiability

証明責務　　領域チェック

- 定義された領域外の部分オペレーター(partial operators)の使用は通常モデル作成者の側のエラーである。
- 次の2種類の構成子に関して自動的にチェックすることが出来ない:
  - 関数が事前条件を持っている場合;
  - 部分オペレーターが適用されたとき。

定義:

```
f:T1 * T2 * … * Tn -> R
f(a1,…,an) == ...
pre ...
```

fの事前条件を次のシグネチュアを持ったブール関数と見なすことが出来る:

```
pre_f:T1 * T2 * … * Tn -> Bool
```

証明責務　　領域チェック

事前条件を持つ関数に対する領域チェック

- もし、関数gが、関数f:T1*…*Tn->Rを、f(a1、…、an)という形で使用する場合: 次のことを示す必要がある。
  - 可能性のある全てのa1,…,an にたいして、次が成立する。

```
pre_f(a1,…,an)
```

例:

```
Delete: Tracker * ContainerId * PhaseId -> Tracker
Delete(tkr,cid,source) ==
   mk_Tracker({cid} <-: tkr.containers,
                Remove(tkr,cid,source).phases)
pre pre_Remove(tkr,cid,source)
```

対象の関数の領域チェックのための証明責務:

```
forall tkr:Tracker,cid:ContainerId,source:PhaseId &
   pre_Delete(tkr,cid,source) =>
      pre_Remove(tkr,cid,source)
```

証明責務　　領域チェック

部分オペレータのための領域チェック

- 部分オペレーターが使用されるたびに、その適用領域内であることを保証しなければならない。。

例:

```
Introduce: Tracker * ContainerId * real * Material ->
   Tracker
Introduce(trk,cid,quan,mat)==
    mk_Tracker(trk.containers munion
                  {cid |-> mk_Container(quan,mat)},
                  trk.phases)
pre cid not in set dom trk.containers
```

munionの適用条件

証明責務は:

```
forall trk:Tracker, cid:ContainerId, quan:real,
   mat:Material &
pre_Introduce(trk,cid,quan,mat) =>
   compatible(trk.containers,
                     {cid |-> mk_Container(quan,mat)})
```

証明責務　　領域チェック

- 部分オペレーターは、下のPermissionでの例のように、事前条件によって、あるいはオペレータの主要部に明示的なチェックを含めることによって適用領域を保証することが可能である。

```
Permission: Tracker * ContainerId * PhaseId -> bool
Permission(mk_Tracker(containers,phases), cid, dest)==
  cid in set dom containers and
  dest in set dom phases and
  card phases(dest).contents < phases(dest).capacity
  and
  containers(cid).material in set
     phases(dest).expected_materials
```

証明責務(太字の関数の領域チェックに必要なもの)

```
forall mk_Tracker(containers,phases):Tracker,
          cid:ContainerId, dest:PhaseId &
  (cid in set dom containers and
      dest in set dom phases) =>
                        dest in set dom phases
```

証明責務　　領域チェック

例: 次の定義において**太字部分**で作成されるべき証明責務は何か。

```
Move: Tracker*ContainerId*PhaseId*PhaseId -> Tracker

Move(trk,cid,ptoid,pfromid) ==

  let pha = mk_Phase(
            trk.phases(ptoid).contents union {cid},
            trk.phases(ptoid).expected_materials,
            trk.phases(ptoid).capacity)  in

  mk_Tracker(trk.containers,
  Remove(trk,cid,pfromid).phases ++ {ptoid |-> pha})

pre Permission(trk,cid,ptoid) and
  pre_Remove(trk,cid,pfromid)

forall trk:Tracker, cid:ContainerId, ptoid:PhaseId &
  pre_Move(trk,cid,ptoid)=>
      ptoid in set dom trk.phases
```

証明責務　　領域チェック

何を、事前条件に含むべきかを決定するのは難しい。

- いくつかの条件は要求仕様によって決定される。
- 多くの条件は、部分オペレーターおよび部分関数の適用時に、領域を保証する必要の為にある。

- 関数定義を書く場合、部分オペレーターを使用する部分をチェックし、そのオペレーションが領域外になることを避けているという保証のため、事前条件をアンド条件で増やして行く必要がある。

証明責務　　満足性(satisfiability)

- 次のように定義された事前条件無しの関数があるとする。

```
f:T1*...*Tn -> R
f(a1,...,an) == ...
```

この関数は、全ての入力に対して、関数の本体の演算結果が正しいタイプである時、***satisfiable*(満足させられた状態)**であるといわれる。形式的にいうと、

```
forall p1:T1,…,pn:Tn & f(p1,…,pn) : R
```

- 事前条件を備えた直接定義関数fの場合

```
f:T1*...*Tn -> R
f(a1,...,an) == ...
```

事前条件を満足する全ての入力に対して、関数の本体の演算結果が正しいタイプであるとき、***satisfiable*(満足させられた状態)**であるといわれる。

```
forall p1:T1,…,pn:Tn &
  pre_f(p1,…,pn) => f(p1,…,pn) : R
```

証明責務　　満足性(satisfiability)

例

```
Introduce:Tracker*ContainerId*real*Material->Tracker
Introduce(trk,cid,quan,mat)==
   mk_Tracker(trk.containers munion
                {cid |-> mk_Container(quan,mat)},
                trk.phases)
pre cid not in set dom trk.containers
```

満足性証明責務(satisfiability proof obligation)は:

```
forall mk_Tracker(containers, phases): Tracker,
      cid: ContainerId, quan: real, mat: Material &
  pre_Introduce(mk_Tracker(containers,phases),
                                  cid,quan, mat)
  =>  Introduce(trk, cid, quan, mat): Tracker
```

証明責務　　満足性(satisfiability)

- satisfiabilityを示す作業の大半は、(集合や、リアル等の)そのタイプの基礎である一般的なタイプが返ってくるということを示すことではなく、そのタイプの不変式を満足していることを示すことに費やされる。

- すべての入力に対して成立することを保証する必要がある。
  → 保証が困難な理由

証明責務　　満足性(satisfiability)

陰定義の関数に関するSatisfiability

関数fが次のように陰定義を用いて定義されたとする。

```
f(a1:T1,…,an:Tn) r:R
pre ...
post ...
```

- この関数は、事前条件を満たす全ての入力に対して、事後条件を満たす正しいタイプを持つ結果が存在するとき、satisfiable（満足させられた状態）であるといわれる。
- 形式的には、

```
forall p1:T1,…,pn:Tn &
   pre_f(p1,…,pn) =>
      exists x:R & post_f(a1,…,an,x)
```

証明責務

満足性(satisfiability)

例:

union type

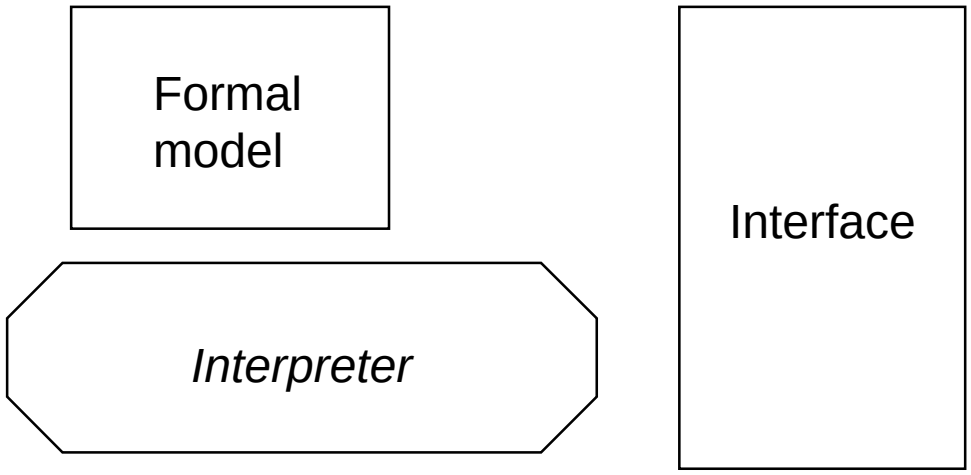
```
Find(trk:Tracker,cid:ContainerId)
                          p:(PhaseId|<NotAllocated>)
pre cid in set dom trk.containers
post if exists pid in set dom trk.phases &
                cid in set trk.phases(pid).contents
     then p in set dom trk.phases and
          cid in set trk.phases(p).contents
     else p = <NotAllocated>
```

満足性証明責務は、次の通りである:

```
forall trk:Tracker, cid:ContainerId &
  pre_Find(trk,cid) =>
      exists p:(PhaseId |<NotAllocated>) &
                              post_Find(trk,cid,p)
```

アニメーション

- 確認（Validation）の目的はモデルが正確に顧客の意図を反映しているという確信を高めることである。
- しかしながら、通常、顧客は、それが形式上でも形式上でなくても、モデル化する言語を理解しないと思ったほうが良い。
- アニメーションは一つのインターフェースを通してのモデルの実行を意味する。
- インターフェースを記述するプログラムとモデルとを結びつけるための動的なリンク機能が存在する限り、インターフェースは任意のプログラミング言語で作成することが出来る。



アニメーション

- (C++で書かれた)インターフェース機能はVDM-SL層に知らされている必要がある。これは、ダイナミックリンクモジュールを用いてこれを行うことが可能である。それは、コンパイルされたC++コードへファイル名を通しての参照機能を与える。例えば:

```
SelectFun: () ->
   <Intro>|<Perm>|<Move>|<Rem>|<Del>|<Stop>
```

これはユーザによってセレクトされたボタンを意味する。

```
GetMove: () -> ContainerId * PhaseId * PhaseId
```

コンテナをフェーズへ指定するためのポップアップ関数を扱う。

```
ShowErr: seq of char -> bool
```

エラー表示をする。

```
ShowTracker: Tracker -> bool
```

追跡システムのオーバービューをアップデートする。

アニメーション

- VDMレベルでは、トップレベルの“Main”関数が、選択および入力を扱うより低いレベルの関数の呼び出しを行う:

```
Main: Tracker -> Tracker
  Main(trk) ==
    cases SelectFun():
      <Intro> -> EvalIntro(trk),
      <Perm>  -> EvalPerm(trk),
      <Move>  -> EvalMove(trk),
      <Rem>   -> EvalRem(trk),
      <Del>   -> EvalDel(trk)
      <Quit>  -> trk
    end

  EvalMove: Tracker -> Tracker
  EvalMove(trk) ==
    let mk_(cid,ptoid,pfromid) = GetMove() in
      if pre_Move(trk,cid,ptoid,pfromid)
      then if ShowTracker(Move(trk,cid,ptoid,pfromid)
           then Main(trk)
           else trk
      else if ShowErr(“Permission for assignment not
   granted”)
           then Main(trk)
           else trk
```

システマティックテスト

- アニメーションを通して得られた確信のレベルは、インターフェースを通ってユーザによって実行された特定のシナリオが上手く行ったと言う事に過ぎない。
- より系統的なテストが可能である:
  - テストケースを作る。
  - 形式的モデル上でテストケースを実行する。
  - 期待する結果と比較する。
- テストケースは、人手または自動的に生成することが出来る。
- しかしながら、自動生成では、莫大な数のテストケースを生成することが出来る。
- 関数プログラムで用いられるテストケースの発生テクニックは、形式的モデルにおいても適用することが出来る。

システマティックテスト

- 次のテストを実行すると、偽という結果が得られた:

```
Permission(mk_Tracker({|->},{|->}),mk_token(1),
   mk_token(2))
```

- 次のテストを行うことによって、Permissionのどの部分が実行されたか(“covered”)が判明する:

```
Permission: Tracker * ContainerId * PhaseId -> bool
Permission(mk_Tracker(containers,phases), cid, dest)
   ==
   cid in set dom containers and
   dest in set dom phases and
   card phases(dest).contents < phases(dest).capacity
   and
   containers(cid).material in set
      phases(dest).expected_materials
```

- テストによって、実行されなかった部分を強調し、他のテストを考えるためにこの情報を利用するツールを使うことが可能である。

証明による確認(validation)

- 系統的な試験およびアニメーションはテストケース、及び使用されたシナリオに関する限り問題ないことを示すに過ぎない。
- 証明によって、一回の分析で、全ての入力のクラスに対して、モデルの振る舞いを評価・保証することが出来る。
- モデルの特性を証明するために、特性を(証明責務のような)論理的な表現として公式化しなければならない。モデルにおいて、保持される必要がある論理表現は、validation conjecturesと呼ばれる。
- 証明は、時間がかかる。機械による支援は非常に制限されている:自動的に一般的な確認推定の証明を構築することができる機械を作ることは不可能である。
- しかし、一度(フォーマルな)証明が作成されると、それをチェックすることができるツールを作ることは可能である。証明を組み立てるには、相当な技術が必要であるが、一度証明されれば、モデルに関する確認において高い保証を与える。

証明による確認(validation)

証明のレベル:

- **教科書レベル**:公式によって支援された自然言語による論述。推論のステップの正当化は人間の洞察(“明らかに…”、“素数の性質によって…”等々)に訴えるやり方。最も読みやすいスタイルであるが、人間でしか正しさのチェックが出来ない。
- **形式的**:別の極端なもの。高度に構造化された論理式の並び。推論の各ステップは、形式的に定義された推論規則によって正当化される。(それぞれの規則は、公理か証明された結果でなければならない。)機械で正しさのチェックが可能。証明するのは非常に困難ではあるが、しかし確度の高い保証(重大なアプリケーションの中で使用される)が得られる。
- **厳密**:定式の高度に構造化したシーケンス、しかしそれらが特定の推論規則ではなく一般的な理論が使用できるように、正当化保証における制限を緩めたもの。

Validation Conjecture

- いかなるフェーズにおいても、そのフェーズに対して正しいタイプのコンテナのみを含んでいる。

```
forall trk: Tracker &
   forall phase in set rng trk.phases &
       forall cid in set phase.contents &
             trk.containers(cid).material in set
                          phase.expected_material
```

教科書タイプの証明(状態不変式)

- Trackerタイプのための不変式は、補助関数MaterialSafeとConsistentの呼び出しを含んでいる。そしてこれら2つの関数が、Validation Conjecture が成り立つことを保証している。

- Trackerタイプが正しく守られると言うことは、その不変式、ひいては Validation conjectureが保持されると言うことである。

```
Tracker :: containers : ContainerInfo
           phases     : PhaseInfo
  inv mk_Tracker(containers,phases) ==
  Consistent(containers,phases) and
  PhasesDistinguished(phases) and
  MaterialSafe(containers,phases)
```

教科書タイプの証明

- MaterialSafeは、正確に最も外側の全称作用素の式(すなわちすべてのTrackerタイプが満たすべきと言う意味)の本体部分を記述するのと同じやり方で定式化されている。

```
MaterialSafe(containers, phases) ==
   forall ph in set rng phases &
       forall cid in set ph.contents &
             cid in set dom containers and
containers(cid).material in set ph.expected_material

forall trk: Tracker &
   forall phase in set rng trk.phases &
       forall cid in set phase.contents &
             trk.containers(cid).material in set
                         phase.expected_material
```

教科書タイプの証明

- Consistentは、フェーズ内のコンテナが、containersのマッピングで既知(パーシャルにならない)であることを保証している。

```
Consistent(containers,phases) ==
   forall ph in set rng phases &
       ph.contents subset dom containers
```

厳密な証明

from trk:Tracker

1 from phase in set rng trk.phases, cid in set phase.contents

1.1 inv_Tracker(trk) Tracker-defn(h)

1.2 Consistent(trk.containers,trk.phases) and-E(Unfold(1.1))

1.3 phase.contents subset dom trk.containers forall-E(Unfold(1,2),1.h1)

1.4 MaterialSafe(trk.containers,trk.phases) and-E(Unfold(1.1)

1.5 cid in set dom trk.containers and

trk.containers(cid).material = phase.expected_material

forall-E(Unfold,1.4,1,h1,1.h2)

infer trk.containers(cid).material = phase.expected_material and-E(1.5)

infer forall phase in set rng trk.phases &

forall cid in set phase.contents &

trk.containers(cid).material =

phase.expected_material forall-forall-I(1)

確認技術の選択

Level of Confidence Required:

Costs:

例えば証明は難しいし、時間もかかるが、テストはかなり自動化できる。

ただし、確認技術のコストと質の関係は時が経つにつれて移り変わる。

まとめ

- 確認:モデルが正確にクライアント要求仕様を反映しているという確信を高める過程。

内部整合性:

  – 領域チェック: 事前条件を持つ、部分オペレーションや、部分関数

  – 直接、陰定義関数のsatisfiability

チェックの正確性:

  アニメーション

  テスト

  証明