社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2005－SE－150 (16)
2005／11／29

# ソフトウェア測定形式化による妥当性検証手法

二木 邦尚† 織田 健‡

†電気通信大学 電気通信学研究科 電子情報学専攻

‡電気通信大学 電気通信学部 情報通信工学科

ソフトウェア品質改善には、品質を定義しそれを評価するソフトウェア測定が必要であるが、それを有効なものとするためには測定データの品質を高める必要がある。データ品質を向上させるには、測定に関わる要素 (モデル、プロセス、結果) に対する妥当性と信頼性を確立する必要がある。本稿では、既存研究において考慮が不十分と考えられる、プロセスと結果に対する妥当性検証を要素を形式化した形で行い、データ品質の向上を図る手法を提案する。また、手法をファンクションポイント測定の一部に適用し、いくつかの効果を確認した。

# A Software Measurement Validation Method with a Formal Approach

Kunitaka Futatsugi† Takeshi Oda‡

†University of Electro Communications Department of Communications and Systems Engineering

‡University of Electro Communications Department of Information and Communication Engineering

A software measurement is necessary to achieve a software quality improvement. It is important for a meaningful measurement to collect high quality data. In order to collect such data validity and reliability about measurement elements (model, process, result) must be established. Many researches have focused on model validity but not taken into consideration processes and results. So in this paper we suggest software measurement validation method which validates measurement processes and results. A formalizing measurement elements in out method makes elements reliable. We have evaluated our method through applying it to software function point measurement.

## 1 はじめに

ソフトウェアの大規模化に伴い、その生産性・品質の低下が問題となっている。ソフトウェアの品質改善を図るためには、品質を定義しそれを評価するソフトウェア測定 [1] が必要になる。測定が有効であるためには、測定データの品質を高める必要があり、そのためには、測定に関わる要素に対する妥当性や信頼性を確立させなければならない。測定に関わる要素には、測定内容や性質を規定する測定モデルや、測定のプロセス、プロセスにより生成される結果群などが存在する。これまで測定の妥当性に関する研究にはいくつかのものが存在するが [2][3][4][5]、測定に含まれる基本的な概念の性質や特定の属性の性質についての妥当性、つまりモデル定義に関するものが主であり、プロセスや結果に対する考慮が不足している。また、測定の信頼性に関する考慮も不十分である。

そこで、本稿では、測定に関わる要素を形式的に捉え、測定プロセスや測定結果についての妥当性を検証することで最終的な測定データの品質向上を図る手法を提案する。また、本手法をファンクションポイントソフトウェア測定法の一部に適用し、本手法により期待される効果の考察を行う。

## 2 測定の妥当性検証

ソフトウェア測定はソフトウェア工学の科学的基礎を提供する存在であり、ソフトウェア開発を工学的に捉えるには必須のものである。測定が有用となるためには測定データの品質が良好であることが求められる。しかし、現状のソフトウェア測定データには誤りが含まれることが多いと言われている [6]。そのようなデータ品質の問題は、ソフトウェア測定の普及を妨げる大きな要因となっている。

測定データの品質を向上させるためには、測定に関わる要素 (**測定要素**と呼ぶ) の妥当性、信頼性を確立する必要がある。測定要素には、データ自体に加え、測定内容や性質を規定する測定モデル、データを生成する過程を規定する測定プロセスが存在する。ここでデータとは、最終的な測定結果だけでなく結果を導くために必要となる中間的な結果も含めたものを指し、本稿では測定の途中結果と最終結果を合わせて**測定プロダクト**と呼ぶ。

測定要素の妥当性を考えるには、それぞれに正し

さの基準を設けそれに関する検査を行なう必要がある。また、信頼性を確立するには測定要素の定義を明確、簡潔で再現性のあるものにし、測定に含まれる曖昧性を排除する必要がある。さらに、測定の一般的な用途の一つに対象の比較検討のためのベンチマークとしての利用が存在し、そのような場合、測定が妥当だと示す枠組み自体にも信頼性が求められる。

以上のような点をふまえ、測定の妥当性検証の枠組みに求められる要件をまとめると、次のようになる。

- 検証範囲に、測定モデル、測定プロセス、測定プロダクトが含まれる
- 検証の枠組みが形式的であり、曖昧性を持たない

これまで測定の妥当性に関する研究を上記要件と照らし合わせてみると、[2][3][4][5] は検証範囲が主にモデル定義部であり、また、検証の枠組みにおける形式性が不十分であると思われる。

そこで本研究では、既存研究で不十分であると思われる測定プロセスと測定プロダクト部分に焦点をあてた妥当性検証を、測定要素を形式化する形で行う方法の提案を行う。

## 3 測定形式化のための枠組み

前節で説明した要件に含まれる形式性を満足させるためには、検証の枠組みに含まれる全ての要素を形式的に表現し、要素の意味を厳密に定義する必要がある。そのような定義には VDM, Z, CafeOBJ 等の形式的仕様記述言語の利用が考えられる。

VDM, Z は集合論を、CafeOBJ は代数をベースとした言語である。要件に含まれる測定プロセスの形式化を実現するには、複数のプロセス毎の結果を保持することが求められ、状態を容易に表現可能な集合論に基づく言語がその記述に適していると思われる。VDM と Z の基本的な表現能力は同じであるとされ [7]、また両言語ともオブジェクト指向の概念を取り入れた拡張言語 (VDM++, Objective-Z) が提案されている。測定妥当性検証の枠組み中で検証される妥当性は複数のものが考えられ、それらには各々を規定する規則が存在する。現在、測定の妥当性を一般的に示すような普遍的な規則は存在していないため [2]、妥当性検証では既存の規則から適当なものを選択して適用したり、個々の測定独自の概念から構成される規則を定義して利用する形態がとられる。そのような場合では、妥当性検証の枠組みの中で、検証の目的に応じて規則を自由に切り替えて適用することが必要となる。オブジェクト指向の概念はシステムの枠組みを変更することなく機能の切り替えを表現するのに適している。

そこで本手法における測定要素の形式化では、オブジェクト指向の概念を含む集合論を基礎とする仕様記述言語を記述言語として用いる。具体的な言語としては今後の実装を踏まえ、ツールの入手性を考慮し VDM++[8] を利用する。

## 4 形式的測定妥当性検証手法

### 4.1 概要

本手法では測定を、測定対象における求めたい属性値を得るための、対象や測定に必要な情報 (**測定情報**と呼ぶ) を入力として持つ関数適用の繰り返しであると捉える。関数値自体も新たな関数の入力となり得るため一つの測定情報となる。測定結果は、測定情報に対するすべての関数適用が終了した時点で生成される。この概念における測定の妥当性は、一つ一つの関数の変換が妥当であること及び、変換した値が測定制約を満たしていることを確かめることで検査される。

ある測定における最初の変換は、実世界の実体から測定に必要な情報を抽出したモデルの生成であり、本手法ではその記述言語に 3 章で説明した仕様記述言語 VDM++ を用いる。モデルはオブジェクト指向におけるクラスで表現し、測定プロダクト及び妥当性検証に必要となる情報を属性として、変換のための関数を操作として記述する。本手法ではこのクラスを**測定クラス**と呼ぶ。測定クラスにおける測定プロセスは、インスタンス化したオブジェクト (**測定オブジェクト**) の状態の更新を引き起こす変換操作の列に相当し、全ての変換が終了した段階におけるオブジェクトの状態変数に測定結果が含まれる。

本稿における測定プロセスの定義は「測定に関連する作業の系列」とし、関連する作業には、測定活動に加え測定妥当性検証活動も含むものとする。そのため測定オブジェクトの状態変数には、測定プロダクトを表現するもの以外に、検証に必要な情報も含み、状態の更新は検証活動によっても発生する。操作への入力である測定情報には検証情報も含むものとする。測定活動のための変換操作を**測定操作**、測定妥当性検証のための操作を**検証操作**と呼ぶ。図 1 に、測定オブジェクトの状態の遷移により測定結果が生成される概念を示す。

測定操作による変換の妥当性は、操作実行の過程で操作が求める制約を入出力が満たしていることを

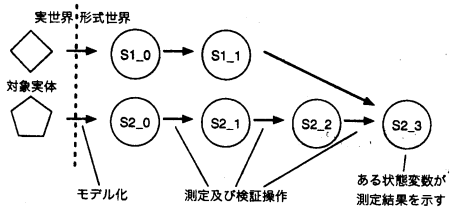


図 1: 測定概念

検査することで行なう。測定操作の制約や検証操作の内容、型に対する制約、他で規定される操作の順序定義を合わせて**測定制約**と呼ぶ。また、測定クラスと測定制約を合わせたものを**測定仕様**と呼ぶ。

本手法における測定の妥当性検証は、測定仕様に対する構文・型検査、未定義性検査などの静的な検証と、測定プロセスの中で実行される測定操作の変換妥当性検査や検証操作による検査などの動的な検証からなる。測定や検証に必要な測定情報の測定者からの入力は、測定仕様を実行する過程で行われる。

測定・検証方法の手順を以下に示す。

1. 測定仕様定義
2. 測定仕様検証
3. 測定仕様実行
4. 測定情報入力
5. 測定情報検証

これらの手順の実行は手法が仮定するいくつかの役割が責任を持つ。手順 1 は役割「測定管理者」が責任を持ち、手順 4 は役割「測定者」が、手順 3, 5 は役割「測定システム」が、手順 2 は、「検証者」及び「測定システム」が責任を持つ。「測定システム」は、妥当性検証の計算や、測定仕様の実行、測定者と情報の受け渡しを担う役割である。「検証者」の存在は、検証の過程に人の介入が必要な場合があることを示している。以上の要素を含む本手法の概念図を図 2 で示す。

続いての節では、本節で示した手法に含まれる個々の概念の詳細を説明する。

### 4.2 測定仕様定義

測定仕様定義では、測定仕様が含む測定クラス、測定順序制約を形式的に定義することで、測定モデル、測定プロセス、測定プロダクト及び、妥当性検証に関する情報や検証内容を形式化する。

測定クラスの定義は、クラス内の型、属性、操作の定義である。型と属性定義は測定対象モデルや測定プロダクト、検証情報のモデル化にあたる。型には型に対する制約を型不変条件として記述する。

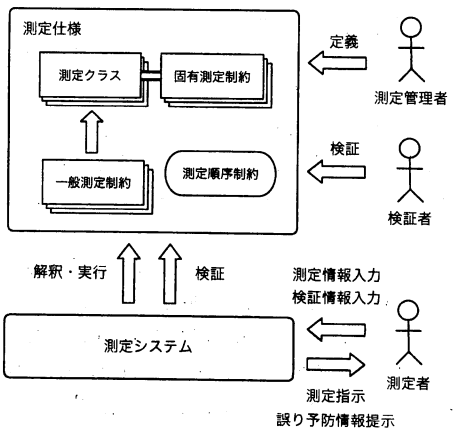


図 2: 手法概要

操作定義は測定操作と検証操作の定義であり、測定内容や検証内容、検証で用いられる測定制約を規定する。測定操作は、測定に必要な情報を測定者から入力として受け取り、それまでの中間的な測定結果を参照し、測定に関連する計算を行い測定オブジェクトの状態を更新する操作である。測定操作には、測定内容の記述に加え、入出力に対する制約を操作の事前条件、事後条件として規定する。

検証操作は、検証に必要な情報を測定者から入力として受け取り、検証に関連する計算を行ない測定オブジェクトの状態を更新する操作である。検証操作には、検証に必要な情報を登録するだけの検証準備のためものや、妥当性検証を行いその真偽を判別する操作が存在する。後者の操作は、検証に関連する属性値を参照しながら操作を実行する。

測定オブジェクトに対する操作メッセージの送信順序は、測定順序制約により記述する。記述形式にはフローチャートを用いる。測定順序制約定義と操作定義により測定プロセスの規定を行う。

型に対する制約、測定操作の制約、検証操作の内容及び、測定順序制約により、測定が満たさなければならない性質を表現する測定制約を規定する。測定制約は、その汎用性や内容により以下の三つに分類される。

- 固有測定制約
- 一般測定制約
- 測定順序制約

固有測定制約は個々の測定固有に含まれる概念に対する制約であり、検証操作の内容や、測定操作の事前・事後条件、属性や属性間の不変条件で表現される。一方、一般測定制約は、ある特徴を持つ測定

集合に適用可能な制約である。個別の測定に依存しない制約であり、制約が想定する性質を持つ測定全てに適用が可能である。一般測定制約の具体的な内容としては、既存の測定妥当性の枠組みで用いられている規則が考えられるが、それらを本手法で採用するには、VDM++ 記述で規則を形式化する必要がある。

測定制約を VDM++ により形式化して記述することにより、妥当性定義の意味を一意に定めることが可能となる。また、測定順序制約を規定したことで、測定プロセスが形式的に規定されたことになる。これにより、測定システムがそれを実行することが可能になり、測定・検証支援の実現につながる。

### 4.3 測定仕様検証

測定仕様検証は、手順 1 で規定した測定仕様に対して静的な検証を行うものである。以下のような検証の種類が存在する。

- 構文検査
- 型検査
- 統合性検査

構文検査では、測定仕様が VDM++ の構文規則に従っていることを検証し、型検査では測定仕様における静的な型の整合性が検査される。

統合性検査では、仕様を実行する段階、つまり、測定プロセスにおける操作の実行の段階おいて、実行時のエラーが発生する場合の有無を検査する。エラー発生条件は、測定クラスの属性に対する不変条件や、測定操作の事前条件及び入力型、結果型に対する不変条件などの測定制約から計算によって求められるため、統合性検査をすることにより、測定プロセスを実施する前段階に測定制約に違反するような非妥当な測定を条件式にて示すことが可能となる。

統合性検査で問題となる条件が判明した場合、測定実施段階での誤りを予防するための対処策を測定管理者が作成する。対処策の例としては、誤り条件そのものや、条件を測定の用語に変換したものを測定者に提示することや、誤りの発生しない入力条件を提示することなどが挙げられる。

### 4.4 測定仕様実行と測定情報入力

測定仕様の静的な検査を終えた段階で、仕様を測定システムが実行する。実行は、測定システムが測定仕様を解釈し、測定順序制約で示される順序で測定クラスの操作を実行する形で行われる。操作の実行過程で、測定情報の入力を測定者に促す。また、4.3 節で説明した統合性検査の結果を調べ、誤り発生条件が存在する場合は、測定管理者による誤り予防作業を実施する。

### 4.5 測定情報検証

測定情報検証は、測定プロダクトを生成するために必要となる測定操作に対する入力値が、測定制約を満たしているかを検査する、動的な妥当性検証である。検証では、入力値に対する型不変条件、操作事前条件、属性不変条件に関する制約が検査される。

## 5 適用例

本章では、本手法のソフトウェア測定への適用例を示し、手法の効果の考察を行う。取り上げる測定法は、ファンクションポイント法であり、その一部分に対する適用を考える。

### 5.1 ファンクションポイント法の概要

ファンクションポイント (以下、FP) 法は、IBM の Albrecht らによって 1970 年代の半ばに開発された、ソフトウェア開発プロジェクトや稼働中のアプリケーションの規模を決定する方法である [9]。FP 値の計算は、アプリケーションを構成する 5 つの主要コンポーネント (外部入力 (EI)、外部出力 (EO)、内部論理ファイル (ILF)、外部インタフェースファイル (EIF)、外部照会 (EQ)) に重み付けした総計値である。

本章で説明する手法の適用範囲は、FP 測定の中の、データファンクション (ILF, EIF) を識別する部分とする。

### 5.2 IFPUG が定める測定規則

測定仕様が含む測定固有制約を規定するためには、測定の満たすべき性質を識別する必要がある。FP におけるそのような性質は、FP の利用を通したソフトウェア開発の効果的な管理を推進することを目的とした非営利組織である IFPUG (International Function Point Users Group)[10] が制定した測定規則から読み取ることができる。今回の測定は、FP 法におけるデータファンクションに関するものであり、それに関連する測定規則を以下に列挙する。

(a) ILF, EIF として識別するデータまたは制御情報のグループは、論理的であり、かつユーザが認識可能でなければならない。

(b) ILF として識別するデータの組は、測定対象アプリケーション境界内の要素処理を通じて保守されなければならない。

(c) EIF として識別されるデータの組は、測定対象アプリケーションから参照され、かつその外部にある必要がある。

(d) EIF として識別されるデータの組は、測定対象アプリケーションでは保守されていてはならない。

(e) EIF として識別されるデータの組は、他のアプリケーションの ILF として保守される必要がある。

上記規則を、測定操作に対する制約や、検証操作の内容として形式化を行う。

### 5.3 測定仕様定義

#### 5.3.1 方針

5.2 節で示した FP の測定規則の形式化方針を、規則と対応させた形にて以下に示す。

(a′) 識別データグループが論理的であることは、それが他のデータファンクションに従属したり帰属していないことにあたり、この制約を、データファンクション間の関係 (継承関係、従属関係) の有無で表現する。ユーザが認識可能であることの確認は、ユーザ認識概念集合に属すことを検査することで行う。ここでの概念とは、データファンクションの候補を表わし、以降ではそれを**エンティティ**と呼ぶ。

(b′) 識別した ILF が要素処理によって保守されるためには、少なくとも一つの EI により処理される必要があり、その制約を、EI との関連を用いて規定する。測定対象アプリケーションが含むトランザクションファンクション集合を管理し、そのうち EI に関して ILF との関連を確認する形で表現する。

(c′) 識別した EIF が、測定対象アプリケーション境界内のトランザクションファンクション集合の要素のうち、データファンクションを保守しないものと関連を持ち、境界内のエンティティ集合に属さないことを確認する。

(d′) 測定対象アプリケーションのトランザクション集合の要素のうち、データファンクションを保守するものが、識別した EIF と関連しないことを確認する。

(e′) アプリケーション毎のエンティティ集合中の ILF と識別されるものを調査し、識別した EIF がそのどれかに含まれることを確認する。

上記述には、今回の手法適用範囲であるデータファンクションの識別以外に、トランザクションファンクションの識別における概念が含まれている。それらに関連する制約の検査については今回行わずその定義の規定にとどめる。また、識別データファンクションがアプリケーション境界内にあることの確認のうち、トランザクションファンクションを経由して検査するものについては、境界内のエンティティ集合を検査することをその代替策とする。

以上を考慮し、測定仕様中に必要な主な操作を以下のように定める。

- 測定操作
  - * エンティティの入力
  - * エンティティのフィールド情報の入力
  - * 対象アプリケーションが境界内に含むエンティティ情報の入力
  - * ILF の識別
  - * EIF の識別
- 検証操作
  - * エンティティ間関係の入力
  - * ユーザ認識情報の入力
  - * ILF に対する制約検証
  - * EIF に対する制約検証

対象アプリケーションのエンティティ情報入力は、データファンクションの識別の前段階として実施されていなければならない、アプリケーション境界の識別を表わしている。

以降では、具体的な測定クラス定義 (型、属性、操作) 内容を示す。

#### 5.3.2 型定義

```
types
 EntityName = seq1 of char;
 FieldName = seq1 of char;
 TransactionName = seq1 of char;
 ApplicationName = seq1 of char;

 EntityRelation = set of (EntityName * EntityName);
 Inheritance = EntityRelation;
 Dependency = EntityRelation;

 UserKnowledge = set of EntityName;

 Field :: f_name : FieldName
          f_type : [FieldType];
 FieldType = <Primal> | <External> | <Implement>;

 Entity :: e_name  : EntityName
           e_type  : [EntityType]
           e_field : [set of Field];
 EntityType = <ILF> | <EIF> | <Implement>;

 Transaction :: t_name   : TransactionName
                t_type   : <EI> | <EO> | <EQ>
                r_entity : set of (EntityName * Maintenance)
 inv mk_Transaction(-,t_t,r_e) == t_t = <EQ> =>
                                    forall r in set r_e &
                                        r.#2 = false;
 Maintenance = bool;

 AllEntity = map ApplicationName to set of Entity;
```

```
AllTransaction = map ApplicationName to set of Transaction
```

仕様中に含まれる要素は全て文字列で識別可能であると仮定する。エンティティ型にはエンティティが認識されるデータファンクションの種類の情報をEntityType型の属性情報として持つ。その属性はオプション型とし、nil値でエンティティの種類が未定義なことを示す。<Implement >はそれが実装上の都合で作られたことを示す。

Field型は、エンティティに含まれるフィールドを表わす。フィールドにはIDに加え性質(主キー、外部キー、実装の都合)の情報も含める。

### 5.3.3 属性定義

```
instance variables
 entities     : AllEntity;
 transactions : AllTransaction;
 inheritance  : Inheritance;
 dependency   : Dependency;
 app_name     : ApplicationName;
 usr_knowledg : UserKnowledge
```

この測定仕様では、属性entitiesが測定結果、途中結果、検証情報を含み、その他の属性は、検証情報を表わしている。

### 5.3.4 測定操作定義

```
operations
 inputEntity : EntityName ==> ()
 inputEntity(e_name) ==
  ( entities := entities ++
     { app_name |-> entities(app_name) union
                    {mk_Entity(e_name,nil,nil) }} )
 pre forall e in set entities(app_name) & e.e_name <> e_name;

 inputField : EntityName * set of Field ==> ()
 inputField(e_name,f_set) == (
  let e in set entities(app_name) be st e.e_name = e_name in
    let e_set = {ent | ent in set entities(app_name) &
                       ent.e_name <> e_name} union
                {mk_Entity(e_name,e.e_type,f_set)} in
        entities := entities ++ {app_name |-> e_set}
 pre (exists e in set entities(app_name) & e.e_name = e_name)
   and (let e in set entities(app_name) be st
        e.e_name = e_name in
         (e.e_field <> nil) => (e.e_field inter f_set = {}));

 inputEntityToApplication:set of Entity*ApplicationName ==>()
 inputEntityToApplication(e_set, appName) ==
  (entities := entities ++ {appName |-> (entities(appName)
       union e_set)})
 pre entities(appName) inter e_set = {};

 identifyILF :  EntityName ==> ()
 identifyILF(e_name) ==(
  let e in set entities(app_name) be st e.e_name = e_name in
     let e_set = {ent | ent in set entities(app_name) &
                        ent.e_name <> e_name} union
                 {mk_Entity(e_name,<ILF>,e.e_field)} in
        entities := entities ++ {app_name |-> e_set})
 pre (exists e in set entities(app_name)& e.e_name = e_name)
   and (forall r in set inheritance & r.#2 <> e_name) and
     (forall r in set dependency & r.#2 <> e_name) and
     (e_name in set usr_knowledge);

 identifyEIF : EntityName ==> ()
 identifyEIF(e_name) ==(
  let e in set entities(app_name) be st e.e_name = e_name in
     let e_set = {ent | ent in set entities(app_name) &
                        ent.e_name <> e_name} union
                 {mk_Entity(e_name,<EIF>,e.e_field)} in
        entities := entities ++ {app_name |-> e_set})
 pre(not exists e in set entities(app_name)&e.e_name=e_name)
    and (forall r in set inheritance & r.#2 <> e_name) and
    (forall r in set dependency & r.#2 <> e_name) and
    (e_name in set usr_knowledge);
```

測定操作は、測定結果や途中結果を導くための操作であるが、操作に含まれる制約により測定制約が規定されるため、測定と同時に妥当性検証も実施される。測定操作実行による検証は、主に固有測定制約についての検証となる。

### 5.3.5 検証操作定義

```
operations
 inputInheritance : EntityName * EntityName ==> ()
 inputInheritance(e_name1,e_name2) ==
  (inheritance := inheritance union {mk_(e_name1,e_name2)})
 pre (mk_(e_name1,e_name2) not in set inheritance) and
     (mk_(e_name1,e_name2) not in set dependency);

 inputDependency : EntityName * EntityName ==> ()
 inputDependency (e_name1,e_name2) == 略

 ILFrelateEI_q : () ==> bool
 ILFrelateEI_q() == (
   return forall e in set entities(app_name) &
            ( e.e_type = <ILF> =>
              exists t in set transactions(app_name) &
              e.e_name in set
                {first | mk_(first,-) in set t.r_entity}));

 EIFrelateTransaction_q : () ==> bool
 EIFrelateTransaction_q() == (
  return forall e in set entities(app_name) &
          (e.e_type = <EIF> =>
          exists t in set transactions(app_name) &
          exists re in set t.r_entity & e.e_name = re.#1));

 EIFnotMaintain_q : () ==> bool
 EIFnotMaintain_q() == (
  return forall e in set entities(app_name) &
          (e.e_type = <EIF> =>
           not (exists t in set transactions(app_name) &
           mk_(e.e_name,true) in set t.r_entity)));

 EIFisILF_q : () ==> bool
 EIFisILF_q() == (
  return (forall e in set entities(app_name) &
         (e.e_type = <EIF>) =>
          exists e_set in set rng({app_name} <-: entities)&
          e in set e_set));

 KeyConstraint_q : () ==> bool
 KeyConstraint_q() == (
  return forall e in set entities(app_name) &
       e.e_field <> nil => forall f in set e.e_field &
        (f.f_type = <External>
         => exists ent in set entities(app_name) &
            exists fi in set ent.e_field &
             fi.f_name = f.f_name and fi.f_type = <Primal>))
```

上記、操作のうち、操作名の末尾が_qのものは、操作で示された妥当性検証を行ないその真偽を返す操作であり、その他は、検証に必要な情報を揃えるための操作である。これら操作が示す制約は、KeyConstraint_q以外FP測定に特化しており固有測定制約である。KeyConstraint_qは関係データベースにおける参照整合性制約を表現しており、その性質はFPから独立しているが操作自体は内部構造に依存しているため、表面上は固有測定制約となっている。

### 5.3.6 測定順序制約定義

図3に測定順序制約のフローチャート表現を示す。図中のend_transactionはトランザクションファンクション測定が終了しているかの確認であり、otherApplication_measuredは対象アプリケーション以外のアプリケーションに対するFP測定が終了しているかの確認である。いずれも本適用例では真とならない。その他の条件は検証操作による検査

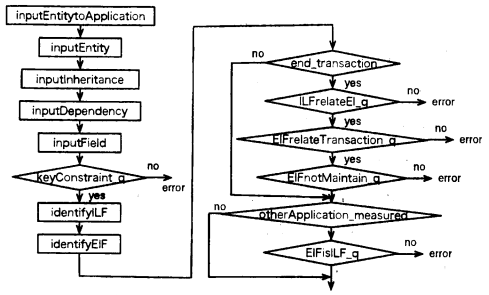


**図 3**: 測定順序制約

を表わし、処理は測定操作の実行を示す。測定操作制約を満たさない場合はエラーとなる。

### 5.4 測定仕様検証

前節で示した測定クラス仕様の静的な部分の検証として、構文、型、統合性の検査を行なう。それらの検証には VDM++ Toolbox Lite[11] を用いる。前節で提示した測定仕様は既にツールを用いて構文・型的な誤りを取り除いた後のものである。統合性検査では、仕様を実行する段階で仕様で未定義な状態に至らないための条件を仕様を計算することで導き出す。今回の仕様では、以下のような条件が示される。

- `app_name in set dom(entities)`
  (エンティティを参照する操作内)
- `is_(ent.e_field,set of Field)`
  (`inputField,keyConstraint` 内)
- `exists e in set entities(app_name) & e.e_name = e_name`
  (`inputField,identifyILF,identifyEIF` 内)

この条件が偽になる場合に、測定や検証作業に問題が発生する可能性が存在する。以下に、そのような状況を測定活動の面で解釈したものと、問題を発生させる場合における対処策を示す。

- 対象アプリケーションが規定されていない
  → データファンクション測定をする前段階に対象アプリケーションを規定しその識別子を一意に定める
- フィールド情報を入力する前段階にフィールド情報が空である (問題無し)
- エンティティに対する付加情報を捉えたが、エンティティが識別されていない
  → (1) あるエンティティの測定に関連する情報を認識する前に、エンティティ自体を一意に識別する値を考える
  → (2) 識別済エンティティの識別子リストを提示する

### 5.5 測定仕様実行と測定情報入力及び測定情報検証

測定システムにより測定仕様を実行し、測定者に測定情報の入力を求める。また、統合性検証作業の結果、誤り予防のための対処策が考えられた場合、測定システムはそれを誤り予防情報として測定者に提示する。システムは入力された情報が測定制約を満たしているかを検証し、そうでない場合それを測定者に示す。

測定システムが未実装なため、本節の内容を実際に確かめることはできない。しかし、いくつかの期待されるシナリオが想定される。例えば、エンティティの属性がエンティティを実装上の都合によるものだと示している時に、それを ILF と識別することや、サブエンティティを ILF と識別することなどの、測定の誤りを検出することが可能である。

### 5.6 適用例での考察

本手法の適用により、ファンクションポイントのデータファンクション測定における測定要素を形式化し、その意味を一意に定めることが可能になる。測定要素の曖昧性を削減することは、測定の信頼性を増加させることにつながる。

また、統合性検証により測定を実施する前段階に測定過程において発生し得る問題を検出可能なことが確認された。問題の発生する条件をもとに誤り予防のための対策をとることにより、非妥当な測定を抑止することが可能になると思われる。今回、統合性検証で検出した誤り条件は、順序に関連しており、測定順序制約が詳細に定めれていた場合、発生しないこともあり得るが、異なる観点からの仕様から類似する概念を抽出したとも捉えられ、測定仕様間の相互補完による品質向上を期待することも可能であると思われる。

本適用例におけるフローチャートで示した測定順序制約は処理の終了条件や繰り返しの必要性などが示されておらず、やや厳密性にかけており、これを解釈し実行することは困難である。また、一般測定制約であるはずのデータベースの参照整合性制約が、測定固有制約と同レベルの操作構造をとっており制約の汎用性を無くしている点も問題である。

## 6 考察

本手法の仕様の静的検証では、仕様の構文・型検査により、測定仕様が VDM++ の構文規則に従い、仕様中の演算子などが VDM++ が規定している有効範囲や型規則を満たしているかを検証することが

できる。これにより、測定仕様の矛盾や曖昧な点を無くし、測定の一意な意味を表現可能となる。また、統合性検証により、測定を実施する前段階において測定過程で発生し得る問題の検出が可能になり、問題発生の抑止につながると思われる。その他の静的検証として、測定仕様の定義自体の正しさを検証することが考えられるが、本手法ではその部分は扱っていない。

測定仕様実行による測定の動的な検証では、測定実施過程で生成される値の妥当性を、測定制約に関する検査を行うことで検証することが可能である。測定は、測定システムが測定順序制約を解釈し測定者に測定情報の入力を求めるため、順序制約が示す範囲の順序的妥当性は保証されるが、現段階における本手法での順序制約には曖昧性が含まれている。

本手法の枠組みにおいて測定の定義自体の検証を行うには、定義レベルでの正しさを形式化し、それと仕様を比較することが必要になると思われる。定義レベルでの正しさを決めるものには、[3][5] 等の既存研究によるものや、測定のスケール型に基づく一般的な性質などがある。それらの中には定義に曖昧性が含まれるものもあり形式化対象となり得る。また、本手法が用いている集合論的な形式化の枠組みの上で新たな定義レベルの正しさを規定することが可能であると考えており、「比率」や「割合」といった単純な複合測定の性質面からの正しさの形式化を検討している。

## 7 おわりに

本稿では、ソフトウェア測定に関連する要素や制約を形式化し、測定プロセスや測定プロセス内で生じる値が測定の満たすべき性質を備えているかを検証することで、最終的な測定結果の品質向上を図る方法の提案を行った。そして、手法をファンクションポイント測定の一部に適用した事例から、測定データの改善につながる、本手法によるいくつかの効果を確認した。

今後の課題としては、測定順序制約の記述形式などの、手法に含まれる曖昧な部分の規定や、妥当性検証範囲の拡張などが挙げられる。

## 参考文献


[1] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics A Rigorous and Practical Approach Second Edition.* PWS Publishing Company, 1997.

[2] Barbara Kitchenham, Shari Lawrence Pfleeger, and Norman Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, 1995.

[3] E.J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.

[4] Norman Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–205, 1994.

[5] Austin C. Melton, David A. Gustafson, James M. Bieman, and Albert L. Baker. A mathematical perspective for software measures research. *Softw. Eng. J.*, 5(5):246–254, 1990.

[6] Victor R. Basili and David M. Weiss. A methodology for collecting valid software engineering data. *IEEE Trans. Software Eng.*, 10(6):728–738, 1984.

[7] I. J. Hayes, C. B. Jones, and J. E. Nicholls. Understanding the differences between vdm and z. *SIGSOFT Softw. Eng. Notes*, 19(3):75–81, 1994.

[8] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems.* Springer, New York, 2005.

[9] デービッド・ヘロン デービッド・ガーマス. **ファンクションポイントの計測と分析**. ピアソン・エデュケーション, 2002. 小泉浩, 中村永, 向井清 訳、児玉公信 監訳.

[10] IFPUG. International function point users group, http://www.ifpug.org.

[11] CSK CORPORATION. VDM++ toolbox user manual -revised for v6.8.3, http://www.vdmbook.com/tools.php.