情報処理学会論文誌 Vol. 49 No. 7 2304–2318 (July 2008)

モデル変換に基づく要求記述を利用した形式仕様の構築

中川博之[†1,*1] 田口研治[†2] 本位田真一[†2,†1]

近年，ソフトウェアの複雑化により，要求分析の重要性が認識されるとともに形式手法によるモデルの詳細化が注目されている．しかし，形式手法による仕様の記述は開発者にとって容易ではなく，要求を満足するソフトウェア構築は依然として容易ではない．そこで本研究では，要求分析法 KAOS により記述された要求記述を形式仕様言語 VDM++の形式仕様へと自動変換する手法と，本手法を用いた開発プロセスを提案する．本研究により，形式仕様の構築が容易になるとともに，要求記述と形式仕様間の整合性が保証され，モデル詳細化によるソフトウェア開発が実現される．

Constructing Formal Specifications from Requirements Specifications Based on Model Transformation

Hiroyuki Nakagawa,[†1,*1] Kenji Taguchi[†2] and Shinichi Honiden[†2,†1]

Requirements analysis and formal methods are techniques for developing complex systems. However, there is little research on reconciling the requirements phase with the formal specification phase. To bridge this gap, we propose a model transformation method which utilizes the KAOS, as a requirement analysis method, into VDM++ formal specifications, and provide developers with a formal specification generator for model transformation. This method and generator enable consistent and effective software development activities.

1. はじめに

近年のソフトウェアの複雑化により，形式手法[1] によるシステムの仕様記述とその開発手法が注目されている．しかし，形式手法によるモデル記述は十分に容易であるとはいえず，設計者が意図しない誤りや，要求に合致しない記述の混入を排除することは難しい．また，現在までに UML のクラス図を利用した形式仕様構築法などが提案されている[4),20)] が，これらの手法を用いたとしてもクラス図と要求との関連が保証されていないため，要求の変更時に形式仕様上での該当する変更箇所を正確に特定することは容易ではない．そこで本研究では，形式仕様の構築を支援するだけでなく，要求と形式仕様間の整合性を確立することを目的とし，要求記述を利用した形式仕様の構築手法と，本手法を用いた開発プロセスを提案する．本論文においては特に，ゴール指向要求分析法の 1 つとして近年注目されている KAOS[2),3)] とその記述モデルを利用し，形式仕様言語 VDM++[4)] の仕様構築手法とその開発プロセスに関して議論する．

本論文では，まず 2 章で形式手法の現状と形式仕様構築の難しさについて論じ，続く 3 章で提案手法である要求分析結果からの形式仕様の自動生成法と本手法を用いた開発プロセスについて説明する．さらに 4 章で提案手法を利用した形式仕様の構築実験とその結果を示し，手法の有効性を検証する．最後に 5 章で提案手法の適用範囲と関連研究について言及した後，6 章でまとめを述べる．

2. 形式手法を用いたソフトウェア開発

形式手法は集合論や論理学など離散数学上の理論を用いてシステムの仕様を記述する開発手法であり，UML[5)] などの図形表現と比較して記述に曖昧さがなく，システムの厳密なモデル化が可能な手法である．また，記述言語の多くが段階的詳細化によるソフトウェア開発を可能としている点も形式手法の特徴である．段階的詳細化とは，その妥当性が確認しやすい抽象的な記述をもとに徐々に記述の内容を具体的なプログラムによる実行形式に近い記述に変換する開発スタイルであり，形式仕様言語を用いることで，詳細化の各段階で正しさ

†1 東京大学
The University of Tokyo
†2 国立情報学研究所
National Institute of Infomatics
∗1 現在，電気通信大学
Presently with The University of Electro - Communications

© 2008 Information Processing Society of Japan

を検証しながらモデルの具体化を進め，そのプロセスで仕様上のバグ混入を排除することができる．

このような形式手法は近年，オブジェクト指向言語への拡張が進み，IBM の Vienna 研究所に端を発する VDM-SL（Vienna Development Methods Specification Language）[6] を拡張した VDM++ [4] などがその代表としてあげられる．VDM++はオブジェクト指向設計に基づいた仕様記述ができる一方で，VDM-SL 同様にどのように機能が実現されるかを記述する明示的記述（explicit specification）と，何が機能として要求されるかを記述する非明示的記述（implicit specification）の 2 つの記述スタイルを持つ段階的詳細化が可能な形式仕様言語である．

しかしながら，VDM++を含めた既存の形式仕様言語は数学的理論を用いるために，開発者にとって記述の難しさや，図形表現と比較してシステムの直感的な把握が難しいといった問題を内包している．たとえ段階的詳細化による開発を実現したとしても，もし最初の仕様記述を構築する段階で要求に合致しない仕様が記述されていたならば，結果として得られる設計モデルや実装モデル（コード）は要求に反したものとなる．たとえば，図書館の書籍を管理するシステムにおける “蔵書を追加する” という機能は，蔵書として書籍のタイトルを扱うのか書籍の実体を扱うのかによりまったく異なる機能となるであろう．

また，近年のシステム開発においては初期段階で要求を確定することが難しく，開発者は要求フェーズ以降も，要求の詳細が明らかになった段階でそれをシステムへ反映させなければならない．さらに，要求の変更に対しても柔軟に対応できなければならない．そのため，要求に矛盾しないシステムを構築するには，要求記述と形式仕様間のトレーサビリティ [14),15)] が確立されている必要がある．図書館システムを例にあげると，当初，同一タイトルの本は 1 冊しか保管しないという前提でシステム設計を進めていた場合に，同一タイトルの本を複数保管したいと要求が変化したとすると，構築中の形式仕様のどの記述を変更すべきかを正確に把握できなければ，結果として要求と形式仕様間に矛盾が生じるであろう．したがって，要求分析結果を基に形式仕様を構築するためには，もし要求記述に機能の詳細が定義されていなければ，形式仕様において記述が完全でない部分を正確に把握しておく必要があり，一方で機能の詳細が定義されている場合は，形式仕様の各記述がどの要求に対応しているかを正確に把握できなければならない．

以上の議論より，本研究では以下に示す形式仕様構築の難しさを扱う．

- **要求に合致する形式仕様の構築：**要求分析結果として獲得された概念を正確に形式仕様へと反映させる難しさ
- **要求の追加・変更への対応：**要求の詳細が要求フェーズ以降で明らかになった場合や要求変更が生じた場合に，その影響範囲を特定し，変更内容を形式仕様の該当箇所へ反映させる難しさ

3. モデル変換による形式仕様の構築

2 章であげた形式仕様構築の難しさを軽減し，要求記述と形式仕様との整合性・対応関係を保証するために，本研究ではゴール指向要求分析法 KAOS により記述された要求記述から形式仕様へのモデル変換法と，本変換手法を利用した開発プロセスを提案する．**図 1** は提案する開発プロセスであり，大きく以下のフェーズにより構成される．

KAOS モデルの構築：開発者はまず，KAOS を用いてシステムに対する要求を分析する．KAOS はゴール指向の要求分析法であり，要求と機能との間に関連を定義することのできる分析法であるとともに，機能をオペレーションとして，また環境上の概念をエンティティとして，システムの基本設計に必要な概念をも記述することのできる手法である．提案手法では，各要求に対応するシステム機能をオペレーションを用いて定義することで，要求記述に相反しないモデルを KAOS 上で構築する．この KAOS モデルにエンティティやオペレーションに関する制約条件を加えることにより，検証条件を含んだ図形表現モデルが完成する．

形式仕様の自動生成：KAOS モデルが完成すると，提案手法を実装したモデル変換ツール

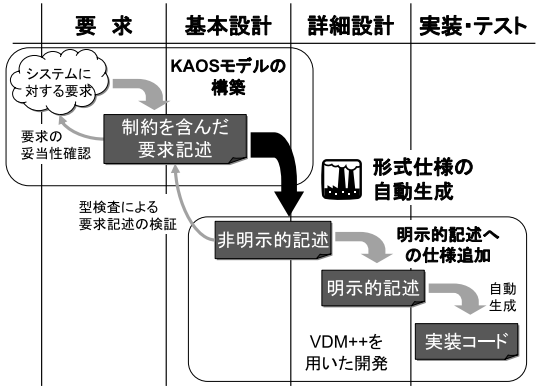

図 1　提案手法による開発プロセス
Fig. 1　Development process based on proposed method.

© 2008 Information Processing Society of Japan

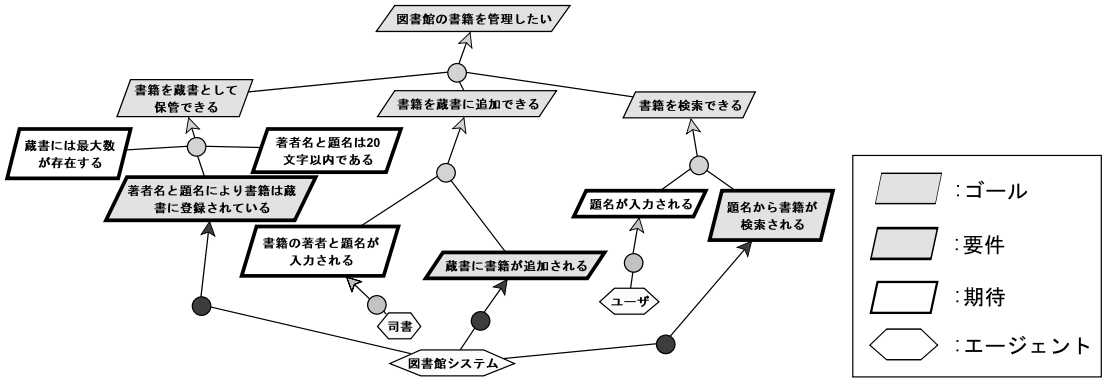

図 2　図書館システムに対するゴールモデル
Fig. 2　Goal model for library system.

により，前プロセスで構築した KAOS モデルを VDM++の非明示的な形式仕様へと変換する．VDM++の非明示的記述は抽象度が高いため，KAOS モデル上に記述された情報のみで自動生成が可能である．この際，KAOS モデルにエンティティやオペレーションに関する制約が記述されていなければ，生成される VDM++記述の該当箇所に仕様追加を促すコメントが付与される．また，生成された非明示的記述が意図された仕様に対して型による誤りなどを含む場合は，VDM++の型検査により判定できる．

明示的記述への仕様追加：開発者は非明示的記述に操作の処理内容などを追加することで，形式仕様を明示的記述へと詳細化する．明示的記述では，すでに KAOS モデルから引き継がれている制約や，設計段階で追加した制約とテストケースを用いることで仕様の正しさを検証する．

以上のプロセスにより，開発者は要求を反映した形式仕様を自動構築できるとともに，提示された該当箇所に仕様を追加することによる詳細化が可能となる．本章では以降，図書館システムの構築を例にあげ，提案する開発プロセスの各フェーズについて説明する．

3.1　KAOS モデルの構築

提案手法では形式仕様との対応関係を定義するために，要求を実現するためのオペレーションも記述可能である KAOS を要求記述に用いる．KAOS は Lamsweerde らによって開発されたゴール分析を基にした要求獲得・分析手法であり[2),3)]，識別したゴールを達成するための要件を系統的に導出できることが特徴としてあげられる．本研究では特に，KAOS 内の各モデル要素に対しての詳細情報付加と構築モデルの XML 形式での取得を実現するために，KAOS のモデリングツールとして知られる Objectiver[12)] をモデル記述に利用する．本節では，まず初めに図書館システム構築の例を用いて KAOS 分析の詳細について説明し，続いて本手法において付加した KAOS モデル記述規則について言及する．

KAOS による要求分析では，まずゴールモデルにおいてシステムが目標とするゴールからシステム要件を抽出する．ゴールはゴールツリーにより，システムが実現すべき要件（Requirement）と，ユーザ・他システムといった外部環境に対する期待（Expectation）へと分割される．**図 2** は図書館システムに対するゴールモデルの一例である．「図書館の書籍を管理したい」という要求は，「書籍を蔵書として保管できる」，「書籍を蔵書に追加できる」，「書籍を検索できる」というサブゴールに分類され，それらはさらに図書館システム，司書，ユーザそれぞれが担当できる責務のレベルにまで詳細化される．また，「著者名と題名は 20 文字以内である」，「蔵書には最大数が存在する」などの書籍・蔵書に関する制約も抽出される．

続いて，KAOS ではシステム構築環境においてモデル化されるべき概念や用語をオブジェクトモデル上にエンティティとして定義する．オブジェクトモデルは，ゴールモデル上に登場する概念を抽出し，それらの関係を記述することで表現される．**図 3** は，図書館システムに対するオブジェクトモデルの一例である．本例では，エンティティとして書籍と蔵書，さらに書籍の著者名や題名といった概念が抽出される．

© 2008 Information Processing Society of Japan

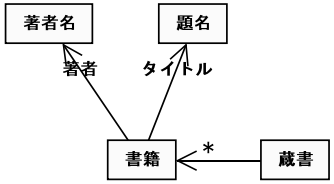

図 3　図書館システムのオブジェクトモデル
Fig. 3　Object model for library system.

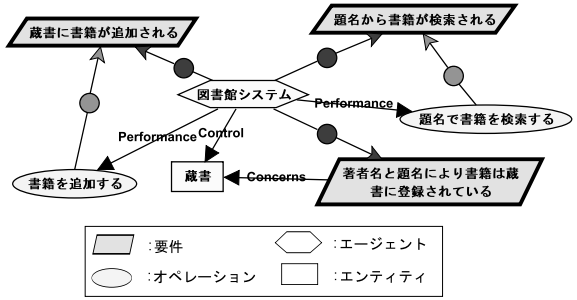

図 4　図書館システムに対する責任モデル
Fig. 4　Responsibility model for library system.

オブジェクトモデルが構築されると，責任モデルにおいてシステムを表現するエージェントごとに要件を集約し，要件を実現するためのオペレーションを定義する．**図 4** は図書館システムに対する責任モデルである．本例では，図書館システムに割り当てられた要件を満たすために「書籍を追加する」,「題名で書籍を検索する」という 2 つのオペレーションが定義される．

KAOS では最後にオペレーションモデルにおいて，オペレーションやエンティティ，イベントを用いてシステムの実行系列を定義する．**図 5** は，図書館システムが持つオペレーション「題名で書籍を検索する」に対するオペレーションモデルである．

以上が通常の KAOS を用いた分析プロセスであり，提案手法でも従来の KAOS 分析プロセスに従って要求を分析するが，KAOS のモデル記述法では図 5 に示すように，オペレーションの入出力となるオブジェクトはエンティティの粒度で記述する必要があり，操作の入力引数や結果として返す値を厳密に表現することができない．そこで提案手法では，形式仕

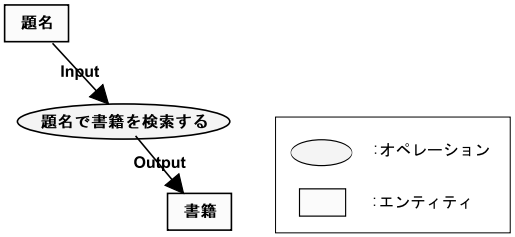

図 5　オペレーションの記述例
Fig. 5　Example of operation description.

様への変換を実現するために KAOS モデル作成時に以下の記述規則を追加する．

提案手法における KAOS モデルの記述規則

- **エンティティの記述：**システム分析で抽出された概念はすべてエンティティとして記述し，必要に応じて制約を付加する．また，VDM++上での写像，タプルといった複合型に対応するエンティティに対しては，VDM++文法に従った型名を制約として記述する[⋆1]．ただし，要素の集合により構成される集合型に関しては KAOS モデル上の関連を用い，要素となるエンティティ側の多重度を“＊”（0 を含む任意の数）とすることで表現する．図 3 の図書館システムの例では，「蔵書」は「書籍」の集合により構成される集合型である．
- **制約の記述：**制約はエンティティ・オペレーションの属性情報として，変換先モデルである VDM++の文法規則に従って記述する[⋆2]．記述された制約は変換手法により，VDM++記述の該当箇所にそれぞれ展開される．本手法が扱う制約は，VDM++におけるオブジェクト，変数，型に対する不変条件と型名，操作に対する事前条件・事後条件である．
- **システム内部変数：**VDM++におけるシステム内部変数は，KAOS モデル上ではエージェントからエンティティに対して “Control” の関係を定義することで表現する．図 4 に記述された「蔵書」エンティティがこれにあたる．

これらの記述規則を付加することで，まずエンティティの記述規則により，KAOS モデル上でオペレーションの入出力に VDM++で扱うすべての概念を指定できるようになる．

⋆1 VDM++における型名は，後継の VDM++による開発時に決定してもよい．
⋆2 提案手法において KAOS モデリングツールとして利用する Objectiver では，エンティティ・オペレーションが複数のプロパティ属性を持つため，これらを制約の記述に利用している．

© 2008 Information Processing Society of Japan

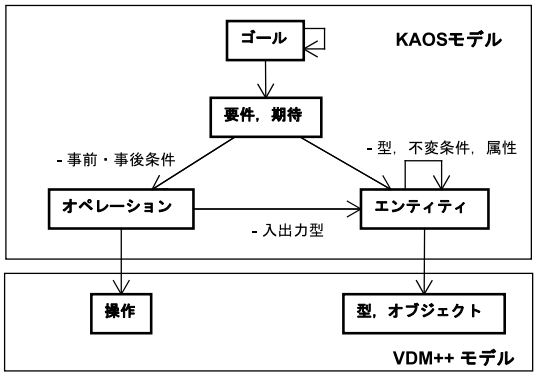

図 6　KAOS と VDM++のモデル間関係
Fig. 6　Relationship between KAOS and VDM++ models.

また，制約の記述とシステム内部変数の記述規則により，システムが直接状態を管理するデータ，つまりシステム内部変数に対しても KAOS のエンティティを利用して固有の制約を記述することが可能になる．さらに，VDM++ではオブジェクトや変数に対する不変条件や，操作に対する事前・事後条件を記述できるが，提案手法では KAOS モデル上のエンティティとオペレーションに対してこれらの制約を付加することで，要求モデルと制約間の関係を明確化することができる．

3.2　形式仕様の自動生成

3.1 節で示した KAOS モデル記述が終わると，KAOS モデルを VDM++の形式仕様へと変換する．本研究では提案するモデル変換手法を Eclipse プラグイン[10] として実装し，形式仕様の自動生成を実現した．この変換ツールは，Objectiver の XML 形式出力モデルを解析し，VDM++の文法に従った形式仕様を出力する．実装した変換アルゴリズムの詳細を付録 A.1 に示す．

形式仕様への変換にあたり，提案手法では KAOS モデルと VDM++モデル間に**図 6** で示す対応関係を定義し，以降のプロセスにより VDM++記述を生成する．

3.2.1　エンティティの判別

提案手法では，まず KAOS モデル上の各エンティティが，オブジェクト，型，集合型，システム内部変数のいずれの概念を表現しているかを判別する．この判別には**表 1** の判別ルールを用い，KAOS モデル内で定義されている他要素との関連から対応する概念を決定する．

表 1　エンティティ判別ルール
Table 1　Discriminating rule for entities.

形式仕様上の概念	判別法
システム内部変数	エージェントから “Control” 線により参照されている
集合型	単一のエンティティのみを参照する関連が定義され，多重度が‘＊’
型	他エンティティを参照していない
オブジェクト	以上の判別を経て残ったエンティティ

図書館システムの例では，「蔵書」エンティティがシステム内部変数かつ集合型，「著者名」，「題名」エンティティが型，「書籍」エンティティがオブジェクトとして抽出される．

3.2.2　各クラスファイルの生成

提案手法では VDM++の形式仕様として，共通定義，オブジェクト定義，システム定義のための 3 種類のクラスファイルを生成する．共通定義クラスは，仕様全体で共有される型を宣言するためのクラスであり，3.2.1 項の判別プロセスで型と判断されたエンティティとその制約情報を記述するクラスである．オブジェクト定義クラスは，判別プロセスにおいてオブジェクトと判別されたエンティティそれぞれに対して生成されるクラスであり，インスタンス変数とその不変条件，コンストラクタ，Getter，Setter により定義される．インスタンス変数に関する情報は関連を利用して取得し，不変条件に関しては KAOS モデルのエンティティに付与された制約情報を参照する．

最後にシステム定義クラスは，構築すべきシステムを定義するためのクラスであり，要件を割り当てられたエージェントに対して定義される．このクラスは内部変数定義部と操作定義部により構成される．内部変数定義部には判別プロセスでシステム内部変数として判別されたエンティティとその制約が記述され，操作定義部にはエージェントに割り当てられたオペレーションの定義（操作名，入出力引数の型，事前・事後条件）が記述される．

図書館システムの例では，共通定義クラスに著者名と題名に関する情報が記述され，オブジェクト定義クラスとして書籍クラスが生成される．また，システム定義クラスとして図書館システムクラスが生成される．**図 7** は変換ツールにより生成された図書館システムクラスの形式仕様である．

3.3　明示的記述への仕様追加

VDM++の非明示的記述が生成されると，開発者はまず構築した KAOS モデルを検証し，その後，得られた記述に情報を付加することで明示的記述へと仕様を具体化する．

© 2008 Information Processing Society of Japan

```
class 図書館システム is subclass of 共通定義
instance variables
private 蔵書 : set of 書籍;
inv
    card 蔵書 <= 1000;

operations
public 書籍を追加する : 書籍 ==> ()
書籍を追加する (a 書籍) == is not yet specified
-- TODO: 本操作の処理内容を記述してください
pre
    a 書籍 not in set 蔵書
post
    a 書籍 in set 蔵書 and card 蔵書 = card 蔵書~ + 1
;

public 題名で書籍を検索する : 題名 ==> 書籍
題名で書籍を検索する (a 題名) == is not yet specified
-- TODO: 本操作の処理内容を記述してください
--pre
-- TODO: 事前条件があれば記述してください
post
    RESULT.タイトルを得る ()  = a 題名
;
end 図書館システム
```

図 7　生成された図書館システムクラスの形式仕様（一部略）
Fig. 7　Generated specification for library system class.

```
class 図書館システム T is subclass of 図書館システム
operations
public t : () ==> seq of bool
t() == return [t1(), t2()];

public t1 : () ==> bool
t1() ==    ...

public t2 : () ==> bool
t2() ==
def
  Book1 = new 書籍 ("Concurrency","Jeff Magee");
  Book2 = new 書籍 ("Design Patterns","Erich Gamma");
  Book3 = new 書籍 ("Design Patterns","Richard Helm");
  a 図書館 = new 図書館システム () in (
  a 図書館.書籍を追加する (Book1);
  a 図書館.書籍を追加する (Book2);
  a 図書館.書籍を追加する (Book3);
return
  a 図書館.題名で書籍を検索する ("Concurrency") = {Book1} and
  a 図書館.題名で書籍を検索する ("Design Patterns")
        = {Book2,Book3}
);
end 図書館システム T
```

図 8　図書館システムに対するテストケース（一部略）
Fig. 8　Test case for library system.

提案する開発プロセスでは，VDM++上への仕様追加の前に，構築した KAOS モデルの妥当性を確認する．ただし，現在 KAOS モデルの妥当性を確認するための手法は確立されていないため，VDM++に変換後のモデル（形式仕様）に対してテストを実施する．本手法では，KAOS モデルから生成される形式仕様とは別に，開発者が別途用意したテストケースを用いることで，意図された仕様[⋆1]を形式仕様が満たすことを確認する．具体的には，VDM++の実行環境である VDM++ Toolbox [13] を利用し，テストケースを用いて型検査をする．もし型検査に対してエラーが発生する場合は，前フェーズである KAOS モデル構築プロセスに戻り，再度要求を分析する．なお，開発者が用意するテストケースは VDM++開発における仕様構築時のテストにも用いる．

これらの型検査，仕様構築時のテストにおいて KAOS モデルの修正が必要になった場合は，本開発プロセスで確立されているトレーサビリティを活用して，修正箇所を追跡することが可能である．これは，提案手法が KAOS と VDM++のモデル要素間に対応関係を定義していることと，KAOS の特性上，KAOS モデル内においても要素間の対応が定義されていることによるものである．

たとえば図書館システムにおける型検査の例では，**図 8** がシステムに対するテストケースの 1 つとすると，生成された仕様記述は操作「題名で書籍を検索する」が同一題名の書籍に対して 1 つの書籍しか返さないので，型チェックエラーを返す．エラーが発生した場合は，開発者は KAOS モデル構築フェーズに戻るが，KAOS モデル上で本操作に対応するオペレーションと関連を持つ要件に着目し，その要件からゴールツリーを上にさかのぼる．も

⋆1 ここでは，テストケースがその意図された仕様を反映していると見なす．

© 2008 Information Processing Society of Japan

し，テストケースが示すとおり複数の書籍を扱うべきであれば，ゴールツリー上に要求[⋆1]として追加し，その要求からエンティティ「書籍」や該当する各操作への関連を定義し，オブジェクトモデルとオペレーションモデルを変更，必要に応じてエンティティ，オペレーションの制約を追加・変更する．提案する開発プロセスではこのような手順で KAOS モデルを更新し，形式仕様を生成することで，KAOS モデルと VDM++モデル間でのトレーサビリティを維持する．

ここで，提案手法における KAOS モデルの妥当性検証法に関して議論する．まず，現状多くの要求分析手法において，それらの記述ツールが検証機能を有していることは少なく，KAOS に関しても同様に検証機能を有しているツールはない．したがって，提案手法では検証機能を有する形式手法用のツール，つまり VDM++ Toolbox により要求記述を検証する．この際，KAOS モデルの型検査用テストケースを VDM++の文法で記述することの難しさを考慮しなければならないが，テストケースでは操作の内部処理に関して考慮する必要はなく，入力に対して期待される結果を記述する程度でよいので，このテストケースの構築は困難ではないと考えられる．さらに，このテストケースはテストファースト[11]の観点から着目すると，VDM++開発プロセスで追加した仕様の正しさを検査するテストデータとしても利用できるため，以降の VDM++を用いた開発にも有効活用できるものである．

テストケースによる型チェックが終わると，VDM++の非明示的記述を明示的記述へと具体化するために形式仕様に情報を追加する．開発者が追加しなければならない必須項目は操作の処理内容である．モデル変換により得られる VDM++の形式仕様は操作の処理内容部が “is not yet specified” といった非明示的記述であるため，開発者は具体的な処理手順に書き換える．この段階で追加された記述は基本的に VDM++形式仕様内でのみ管理する．制約条件をリファインメントした結果として追加された制約条件に関しても，通常は KAOS モデルに記載する粒度よりは詳細な条件であると考えられることから，KAOS モデルに記載している制約条件は更新しない．ただし，もし追加された制約が KAOS モデルの妥当性に影響を与えるのであれば，それは要求に矛盾したものであり，その制約を追加すべきであるか検討し，それでもなお必要な制約であれば KAOS モデルの再検討へ移る．

明示的記述が完成すると，テストケースを再度利用して，操作の処理が正確に記述されていることを確認する．

⋆1 たとえば，「検索結果は複数同時に表示される」などが考えられる．

3.4　要求変更への対応

最後に要求の変化に対する提案手法のアプローチを述べる．提案手法では，KAOS モデルと VDM++モデルとの間に図 6 の対応関係を定義することで，形式仕様の自動生成だけでなく，要求変更時の形式仕様上の変更が必要な箇所の追跡を可能にしている．

要求の変化は，KAOS ゴールモデルにおけるゴールの変化に対応するが，まずゴールの変更に影響する要件をゴール・要件間の関連線により抽出することで，システムに対する要件の変更を把握することができる．この要件の変更に対しては，もし影響を受ける要件とエンティティ間に関連線が存在すれば，該当エンティティの必要性・十分性，型，不変条件，他エンティティとの関連情報を再検討する必要があり，もしオペレーションに対して関連が存在すれば，そのオペレーションの必要性・十分性，入出力型，事前・事後条件を再検討する必要がある．

このように提案手法では，KAOS モデル上で要求変更への対応を記述することで，結果として VDM++上においても変更が必要な箇所を特定することができるとともに，要求変更に対応した VDM++の非明示的記述による形式仕様も自動的に構築することができる．

4. 実　　験

提案する形式仕様生成法を利用した開発プロセスの有用性と，変換手法の妥当性および有用性を検証するために，提案手法を Eclipse プラグインとして実装した変換ツールを用いた形式仕様の構築実験を実施した．本章では実験内容とその結果を示し，提案手法の有効性を評価する．

4.1　実 験 概 要

本研究において実施した実験は大きく 2 種類に分類される．まず，「要求の追加・変更への対応」の難しさに対して，本研究で提案する要求のトレーサビリティが有効に機能することを評価するために，要求記述から形式仕様への追跡プロセスを確認する実験を実施した（実験 1）．具体的には，すでに構築された要求記述に対して要求変更を加え，その要求変更がどのように形式仕様の該当箇所に反映されるかを，変更前後の VDM++形式仕様の変化（変更箇所）を計測することで確認した．

続いて，「要求に合致する形式仕様の構築」の難しさに対して提案手法の有効性を評価するために，KAOS により記述された要求記述を基に提案手法が生成する形式仕様がどの程度 VDM++による開発に利用可能であるかを評価した（実験 2）．具体的には，2 つのサンプルシステムに対して KAOS モデルを構築し，生成された形式仕様に被験者がどの程度の

© 2008 Information Processing Society of Japan

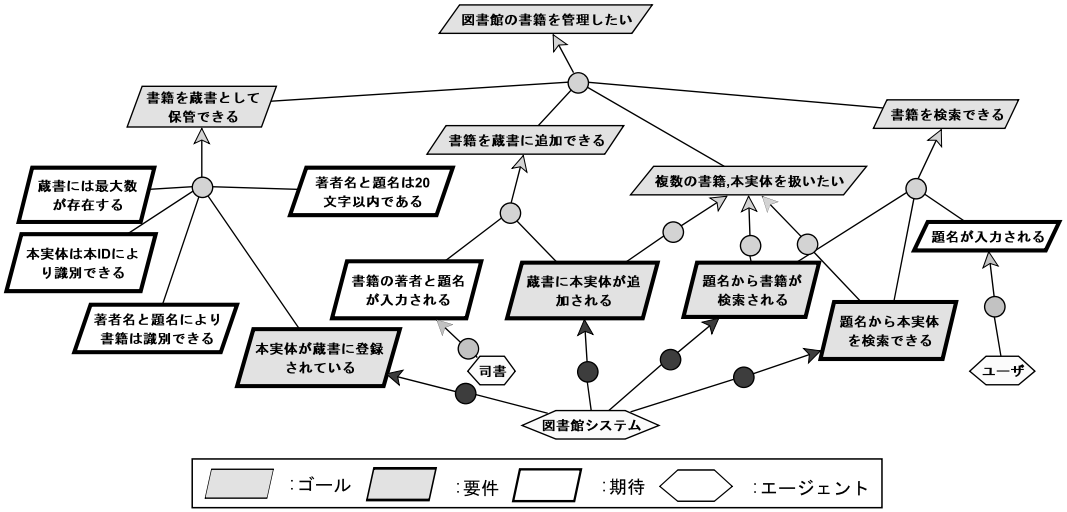

図 9　要求変更後のゴールモデル
Fig. 9　Goal model after requirements changed.

情報を追加することで明示的記述，つまり VDM++ Toolbox で実行可能な形式仕様を構築するかを追加・変更箇所とそのサイズ（行数）の計測により評価した．

実験 1，2 ともに，著者のうち 1 名がシステムに対する要求・機能を提示し，別の 1 名が KAOS モデルの構築および VDM++の明示的記述の構築を担当する形態で実験を実施した．

4.2 実　験　1

はじめに，3 章で示した図書館システムに対して要求を追加・変更することで，KAOS モデルと生成される VDM++の形式仕様の変化を計測した．本実験で新たに追加した要求は以下の 3 つである．

- **R1：**図書館に同一タイトルの書籍を複数冊保管できるようにしたい．
- **R2：**蔵書の追加時には一度に複数の書籍を登録できるようにしたい．
- **R3：**蔵書の検索結果は複数表示してほしい．

要求の変更にはシステムが対象とする環境のモデリングにも影響を与えるものと，システムの振舞いを変化させるだけで対応可能なものの 2 種類があるが，R1 は環境モデリングに対しても影響を与える要求変更であり，R2 はシステムの振舞いを変化させることにより対応が可能な要求変更である．また，R3 は 3.3 節で生じた型チェックエラーの原因となった暗黙的な要求を明示したものであり，要求フェーズ以降で明らかになった要求として分類できる．

まず，R1〜R3 の要求に対応できるように 3 章で示した KAOS モデルを変更した．変更後のゴールモデルを**図 9**，オブジェクトモデルを**図 10** に，KAOS モデル内での全変更箇所を**図 11** に示す．

続いて，要求変更後の KAOS モデルに対して VDM++の形式仕様を再生成し，変更前の記述と比較した．**表 2** は KAOS モデルの変更前後における形式仕様の変更箇所を示したものである．主な相違点として，まず，共通定義クラスに「識別子」型に関する型宣言が新たに追加され，オブジェクトクラスとして「本実体」クラスが新たに生成された点があげられる．これらは環境モデルの変化により生じた形式仕様上の変更といえる．

一方，図書館システムクラスに関しては，まず本実体を検索する操作が追加され，KAOS のオペレーションモデルに記述された入出力型と事後条件に関する記述もあわせて追加された．事前条件は KAOS モデルに記述されていなかったため，記述を促すコメントが該当箇所に挿入された．また，KAOS モデル上において変更を加えられた既存の操作に関しても，操作名，入力引数，事前・事後条件の該当箇所が KAOS モデル上で与えた変更に対して過

© 2008 Information Processing Society of Japan

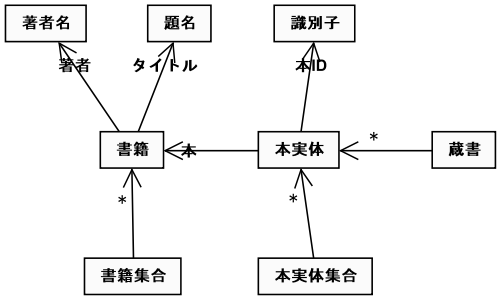

図 10　要求変更後のオブジェクトモデル
Fig. 10　Object model after requirements changed.

不足なく変更が反映されていることが確認できた．

以上の結果より，R1，R2，R3 の変更が，KAOS モデルの各要素を通じて形式仕様上の該当箇所へと変更を与えていることが分かった．なお，R2，R3 は形式仕様上では図書館システムクラスの操作のみに影響を与えているが，KAOS モデル上ではオブジェクトモデルに対する変更も発生させている．これは，要求の追加・変更により最終的に形式仕様へと変化を与えない概念上の変化も抽出，記録しておくことが可能であることを示しており，要求記述と形式仕様間でのトレーサビリティを確立するために必要な情報である．

4.3 実　験　2

次に，図書館システムと宅配便追跡システムの 2 つのサンプルシステムを対象として，KAOS 分析結果から生成された非明示的記述に対してどの程度の情報を追加することで明示的記述，つまり VDM++ Toolbox で実行可能な形式仕様となるかを確認し，追加情報量とその構成内容を評価した．実験では，まず被験者が KAOS モデルを構築し，変更ツールにより VDM++仕様を生成後，被験者が仕様を追加して明示的記述を構築した．ただし，本実験では単に操作部の処理内容を記述するだけでなく，被験者は自身が形式仕様を構築すると想定して，生成された記述部分も必要に応じて変更した．また，本来再利用箇所ごとの情報量，構築コストを考慮した評価をすべきであるが，その計測が難しいため，本実験では追加・変更箇所とその行数により手法の有効性を評価した．特に利用率に関しては，カバー率

$$\text{カバー率} = \frac{\text{明示的記述の行数} - \text{追加・変更行数}}{\text{明示的記述の行数}}$$

ゴールモデルの変更点
- **G1:** 要件「著者名と題名により書籍は蔵書に登録されている」を「本実体が蔵書に登録されている」へ変更し，期待「著者名と題名により書籍は識別できる」を追加（← R1）
- **G2:** 期待「本実体は本 ID により識別できる」を追加（← R1）
- **G3:** 要件「蔵書に書籍が追加される」を「蔵書に本実体が追加される」へ変更（← R1）
- **G4:** サブゴール「複数の書籍，本実体を扱いたい」を追加（← R2, R3）
- **G5:** 「題名から本実体を検索できる」を追加（← R1）

オブジェクトモデルの変更点
- **Ob1:** 「本実体」を追加し，蔵書の管理対象とした（← G1,G3, G5）
- **Ob2:** 「識別子」型をエンティティとして追加（← G2）
- **Ob3:** 識別子型を持つ「本 ID」と書籍型である「本」を定義（← G1,G2）
- **Ob4:** 書籍，本実体に対してそれぞれの集合型「書籍集合」，「本実体集合」を定義（← G4）

責任モデルの変更点
- **Resp1:** オペレーション「題名で本実体を検索する」を新たに定義（← G5）
- **Resp2:** オペレーション「書籍を追加する」を「本実体を追加する」へと変更（← G3）

オペレーションモデルの変更点
- **Op1:** オペレーション「本実体を追加する」の入出力エンティティと制約（事前・事後条件）を変更（← Ob1,Ob4,Resp2）
- **Op2:** オペレーション「題名で書籍を検索する」の出力エンティティを「書籍」から「書籍集合」へ変更（← Ob4）
- **Op3:** オペレーション「題名で本実体を検索する」の入出力エンティティと制約（事後条件）を定義（← Ob1,Ob4,Resp1）

図 11　KAOS モデルの変更箇所（括弧内は変更要因）
Fig. 11　Alterations in KAOS model.

を計測することで，変換ツールにより生成された形式仕様が最終的に被験者により構築された明示的記述においてどの程度利用されているかを評価した．

4.3.1 図書館システム

実験対象とした図書館システムは実験 1 で追加した要求も考慮したシステムであり，形式仕様の生成には実験 1 で構築した KAOS モデルを用いた．図書館システムに対する明示的記述の構築実験結果を**表 3** に示す．生成された形式仕様に対して被験者は，本実体クラス

© 2008 Information Processing Society of Japan

表 2　要求の変化にともなう形式仕様の変更箇所
Table 2　Alterations of VDM++ specification by requirements changes.

クラス名	追加・変更箇所
共通定義	・本実体の ID として利用する「識別子」型が生成（R1）
書籍	追加・変更無し
本実体	・管理対象が「本実体」へと変化したことにより生成（R1）
図書館システム	・システム内部変数である「蔵書」の型が「本実体」集合へ変化（R1） ・操作「書籍を追加する」が「本実体を追加する」へ 変化（R1） ・操作「本実体を追加する」の入力型が「本実体」集合となり，事前・事後条件も変化（R1，R2） ・操作「題名で書籍を検索する」の出力型，事後条件が変化（R3） ・操作「題名で本実体を検索する」が生成（R1）

表 3　図書館システムに対する形式仕様の構築
Table 3　Result of acquiring formal specification for library system.

生成クラス	生成行数	追加変更行数	カバー率	追加変更項目
共通定義	19	0	1.000	
書籍	36	0	1.000	
本実体	36	3	0.923	簡便な Getter の追加
図書館システム	40	7	0.825	初期値・操作処理内容の追加
合計	131	10	0.925	

の簡便な Getter [⋆1] と，図書館システムクラスにおける内部変数の初期値，操作の処理内容を追加した．

このうち非明示的記述として記述されるべき項目は簡便な Getter の定義部分のみであり，その他の項目は開発者が追加・検討すべき項目である．

4.3.2　宅配便追跡システム

宅配便追跡システムとは，宅配便利用者に対して荷物の現在位置を追跡・情報提供するシステムであり，機能（操作）として荷物の現在地確認や，荷物の DB への登録，現在地の変更，荷物の検索，基準の最大配送日を超えている配送異常のチェックの 5 つの機能が求められているシステムである．**図 12** は宅配便追跡システムに対する概念を分析したオブジェクトモデルである．

宅配便追跡システムに対する実験結果を**表 4** に示す．被験者が追加した情報としては，まず，共通定義クラスにおける写像の初期値と共通関数があげられる．共通関数とはすべてのクラスで利用可能な関数であり，本実験では日付の比較や，日数計算といった日付に対する関数が定義された．オブジェクトである荷物クラスに関しては，本クラスが持つべきメソッドが追加され，コンストラクタの引数が変更された．後者は，変換ツールがインスタンス変数をすべて引数としたコンストラクタを生成するため，オブジェクト生成時に必要と思われる引数のみを被験者が選択したことによる変更である．システムクラスの追加内容は，図書館システム同様，インスタンス変数の初期値と操作の処理内容であった．以上，システム全体における自動生成記述のカバー率は，システムの操作や共通関数の処理内容記述量が多く，約 80%であった．

⋆1 ここでの簡便な Getter とは，インスタンス変数を単純に返すものではなく，他オブジェクトへのアクセスや変数処理を経た結果を返す Getter を指す．

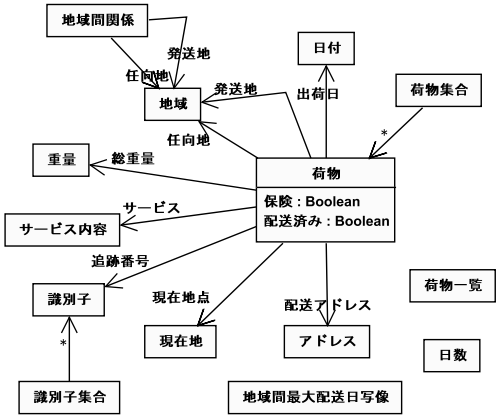

図 12　宅配便追跡システムのオブジェクトモデル
Fig. 12　Object model for delivery tracking system.

表 4　宅配便追跡システムに対する形式仕様の構築
Table 4　Result of acquiring formal specification for tracking system.

生成クラス	生成行数	追加変更行数	カバー率	追加変更項目
共通定義	44	22	0.662	初期値の追加，共通関数を定義
地域間関係	36	0	1.000	
荷物	104	21	0.822	コンストラクタ引数整理，操作の追加，初期値の追加，Getter の名称変更
宅配便追跡システム	87	23	0.747	初期値・操作処理内容の追加
合計	271	66	0.787	

© 2008 Information Processing Society of Japan

表 5　形式仕様の追加・変更項目（図中の数値は追加・変更行数）
Table 5　Breakdown list of changes for two systems' specifications.

No.	項目	図書館	宅配便
1	Getter 定義の追加・名称変更	2	2
2	コンストラクタ引数の整理	—	2
3	オブジェクトメソッド定義の追加	—	2
4	初期値の追加	1	8
5	Getter 処理内容の追加	1	2
6	コンストラクタ処理内容の変更	—	2
7	オブジェクトメソッド処理内容の追加	—	9
8	共通関数の追加	—	17
9	操作処理内容の追加	6	22
合計		10	66

4.3.3　追加・変更項目に関する議論

ここで，実験 2 で扱った 2 つのシステムに対して被験者が仕様を追加・変更した内容に関して議論する．**表 5** は被験者が追加・変更した仕様の内容を分類し，それぞれの行数を集計したものである．このうち No.1〜No.3 が非明示的記述として記述されるべき仕様，No.4〜No.9 が明示的記述構築時に開発者が追加すべき仕様と考えられる．

非明示的記述として記述されるべき仕様に関しては，Getter 定義の追加・変更（No.1）やコンストラクタ引数の整理（No.2）が開発者の判断で加えなければならない追加・変更作業である．ただし，これらは自動生成されている Getter，コンストラクタを活用して実施することができるものであり，提案手法により開発者の作業は軽減されるものと考えられる．一方，オブジェクトのメソッド定義（No.3）に関しては，KAOS モデル上でエンティティに対してメソッドを定義できないことから，現在の提案手法では開発者が手作業で記述を追加する必要がある．これは KAOS の記述能力に起因した本手法の現時点での限界といえよう．

明示的記述として記述されるべき仕様に対しては，コンストラクタ（No.7）やオブジェクトメソッドの処理内容（No.6）が非明示的記述での追加・変更作業に依存して必要となる仕様であり，初期値（No.4）や共通関数（No.8）が構築システムによって必要となる仕様である．開発者が必ず検討・追加しなければならない仕様は，操作処理内容（No.9）である．

以上の追加・変更項目を総括すると，被験者が追加した仕様の大部分が明示的記述構築時の追加記述に分類されるオブジェクトメソッドや操作の処理内容，共通関数の記述であり，操作に関しては入出力型，事前・事後条件がすでに形式仕様中に記述されていることから，提案手法を利用した明示的記述の構築は従来の形式仕様構築における段階的詳細化プロセスとほぼ合致しているといえよう．また，自動生成される形式仕様は開発者の意図を完全に反映した非明示的記述ではないが，開発者の意図を反映した非明示的記述および明示的記述の構築を支援し，有効利用できるものと考えられる．

4.4　考　　察

実験結果をもとに 2 章で定義した形式仕様構築の難しさに対する本手法の有効性を評価する．

4.4.1　要求の追加・変更への対応

実験 1 では，再構築前後の KAOS モデルを比較することで，要求の変化が図 6 のモデル間関係に従って KAOS モデル内で伝播し，結果として形式仕様の該当部分のみへ変化を与えていることが分かった．このように，本手法では要求の変化時，要求の追加定義時にも KAOS モデルを再構築することで，要求に矛盾しない VDM++仕様の再構築が可能である．この結果は 3.3 節で示した逆方向の追跡プロセスとあわせて，要求記述である KAOS モデルと形式仕様である VDM++モデル間のトレーサビリティを確立していることを示すものである．今後，VDM++仕様上で設計者が加えた追加・変更内容の管理機能をツール実装し，開発プロセスに導入することで，繰返しを前提としたイテレーティブな開発にも適用できるものと考える．

加えて，形式仕様に限らず，通常正しいモデルを一度で正確に記述するのは難しい．本実験でも要求変更時に，新たに追加したエンティティに対する型情報の記述忘れや操作の変更漏れがあったが，これらは VDM++ ToolBox での型検査により検出することができた．提案手法ではテスティングによる KAOS モデルの検証を実現しているが，これは要求の変更を確実に形式仕様へ反映させるという意味でも有効であるといえる．

4.4.2　要求に合致する形式仕様の構築

実験 2 では，提案手法により自動生成される形式仕様が VDM++開発における明示的記述の構築に有効利用できることが確認できた．特に，表 5 で示した追加・変更内容から，生成される形式仕様は開発者が構築する非明示的記述に近く，たとえ開発者が非明示的記述としての仕様を追加したとしても，本プロセスにおける矛盾混入の可能性は軽減できると考えられる．これは，VDM++の形式仕様と KAOS モデルとの重複部分が KAOS モデルの記述へと統一された結果によるものといえよう．また，明示的記述を構築するために開発者が仕様を追加しなければならない操作処理内容に関しても，操作の入出力型と事前・事後条件がすでに KAOS モデル構築時に検討されているため，開発者の仕様追加による矛盾混入

© 2008 Information Processing Society of Japan

も避けられる．ただし，オブジェクトのメソッド定義が現在の変換手法では生成できないため，要求に対する厳密な矛盾の排除にはこれらに対する要求記述上での記述方法を検討する必要がある．この問題に対しては，新たな変換手法を検討するとともに，KAOS や他の要求分析法の発展に期待したい．

形式仕様構築のプロセスに着目すると，提案する開発プロセスでは，KAOS モデルの図形要素を活用し，系統だった手順に従った分析を進めることで VDM++仕様のテンプレートが生成され，テストケースを利用した要求記述，形式仕様の検証が可能である．これは特に形式手法に不慣れな開発者にとって，記述の手順を明確化するという観点と，誤りの箇所を特定するという観点から要求分析結果を正確に形式仕様へと反映させるための有効な開発プロセスであると考えられる．

5. 適用範囲と関連研究

最後に本手法の適用範囲と関連研究について論じる．本手法では基本設計モデルを重複部分として，要求記述と基本設計モデルとの連携に KAOS を，基本設計モデルと詳細設計モデルとの連携に VDM++を利用することでモデル間の連携を実現している．

KAOS はシステムの要求を定義するゴールモデルと，概念間の関係を定義するオブジェクトモデル，システムの動的側面を記述するオペレーションモデルの提供により要求モデルとシステム基本設計モデル要素間の対応付けを実現している．本手法でもこの対応関係を図 6 に示す形態で，モデル変換およびトレーサビリティの確立に利用している．KAOS と同じゴール指向要求分析法の代表的な手法として i* [17] があるが，i*はステークホルダ間の要求を分析するといった初期の要求フェーズの活動に重点を置いた手法であり，要求とエンティティやオペレーションの関係まで厳密に定義することができない．これは，ユースケース[16] に関しても同様である．提案手法を適用する場合は，基本設計モデルの要素と要求とを対応付けられる要求記述法の利用が前提となる．我々はマルチエージェントシステムの分析モデル構築法として要求記述からのモデル変換法[24),25)] を提案しているが，この手法においても同様の観点から KAOS を拡張利用している．

一方，VDM++に関しては，提案手法では概要設計モデルとして非明示的記述を，詳細設計モデルとして明示的記述を対応付けている．記述の抽象度に関しては，多くの形式仕様言語がこの 2 つのモデル記述を可能としている．しかしながら仕様をツールによりテスト，実行できるという観点から，本研究では VDM++を形式仕様記述言語として選択した．たとえば Z [7] では基本的には記述は非明示的であり，すべてを実行することはできない．また，B メソッド[8] では初期仕様の正しさを確認することが困難である[18]．

VDM-SL ではなく VDM++を用いたのは，KAOS や UML クラス図などのオブジェクト指向モデリングとの親和性を考慮した結果である．要求のトレーサビリティを向上させるという観点からも，オブジェクト指向の形式仕様言語が有効であると考える．形式仕様言語のオブジェクト指向への拡張という点では Object-Z [9] などをあげることができるが，これらはいずれもツールサポートが十分ではない．

形式仕様構築の自動化および支援に関する関連研究としては，図形表現モデルから形式仕様への変換に関する研究として，特に UML クラス図からの変換手法が多く提案されている．Dupuy らは，UML 記述ツールとして知られる Rational Rose ツール[19] で記述されたクラス図と制約を表現するアノテーションから形式仕様言語である Z へ変換するツール RoZ [20] を提案している．また，Snook は，クラス図と状態遷移図から B の形式仕様へと変換するツール[21] を提案している．これらは直接形式仕様を記述しないという点で，形式仕様言語によるモデル化を容易にしているが，クラス図の記述に要求が反映されていることが前提となる．また，要求の変化を考慮した場合に，クラス図や制約から該当箇所を特定することは困難であり，要求の変化への対応が難しいと考えられる．

Fitzgerald らは特にオブジェクト指向開発をターゲットとして，クラス図を用いた VDM++形式仕様の構築法と形式手法を利用したソフトウェア開発法を提唱している[4),⋆1]が，要求記述と形式仕様間の関係は明確には定義していない．

最後に，定理証明による検証にはコストがかかることと，定式的証明は必ずしも操作的な正しさを保証するものではないことから，形式手法の分野においても近年テスティングが注目され始めている[22]．VDM++に関しても，Beck らの Java 単体テスト・フレームワーク JUnit [23] と同等の VDMUnit [4] が提案されている．我々が提案する開発プロセスにも当分野の研究成果を取り入れることで，テスティングのさらなる効率化が期待できる．

6. ま と め

本研究では，ソフトウェア開発における形式仕様構築の困難さと，要求記述と形式仕様間のギャップを解消するために，ゴール指向要求分析法 KAOS を利用した VDM++の形式仕様生成法と，本手法を利用した開発プロセスを提案した．また，提案手法を実現するモデル変換ツールを実装し，要求変更実験および明示的記述の構築実験を実施することで提案手法

⋆1 本機能は VDM++ Toolbox において，Rational Rose ツールとの連携により実現されている．

© 2008 Information Processing Society of Japan

の有効性を評価した．

今後は，要求変更により変更されたKAOSモデルとVDM++形式仕様間の整合性を確実に保証するために，双方間で生じる矛盾の検出機能を付与する予定である．また，KAOSのゴールモデルにおけるゴール間の論理構造や，各ゴールに記述された時相論理式の有効利用法についても議論したい．加えて，提案手法は形式仕様を自動生成する点で有効であるものの，現段階ではテストケース作成に関する支援が提供されていない．テストケースの作成に関してもVDM++の文法知識が要求されることや，多くのテストケースを作成するのは煩雑な作業であることから，テストケースの作成に関してもKAOSやUMLなどの図形表現モデルを利用したテストケースの自動生成機能を追加したい．本研究の試みが，複雑化するソフトウェア構築の有効な手段として，形式手法を普及させる一助となれば幸いである．

参 考 文 献

1) 荒木啓二郎，張　漢明：プログラム仕様記述論，オーム社 (2002).
2) Dardenne, A., van Lamsweerde, A. and Fickas, S.: Goal-Directed Requirements Acquisition, *Science of Computer Programming*, Vol.20, 1993, pp.3–50 (1993).
3) Leiter, E.: Reasoning about Agents in Goal-oriented Requirements Engineering, Ph.D. thesis, Universite Catholique de Louvain (2001).
4) Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N. and Verhoef, M.: *Validated Designs for Object-oriented Systems*, Springer (2005).
5) Object Management Group: Unified Modeling Language. http://www.uml.org/
6) Jones, C.B.: *Systematic Software Development using VDM, 2nd edition*, Prentice-Hall (1994).
7) Woodcock, J. and Davies, J.: *Using Z*, Prentice Hall (1996).
8) Abrial, J.R.: *The B-Book*, Cambridge Univ Press (1996).
9) Smith, G.: *The Object-Z Specification Language, Advances in Formal Methods*, Kluwer Academic Publishers (2000).
10) Eclipse Plugin Central. http://www.eclipseplugincentral.com/
11) Martin, R.C.: *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall (2002). 瀬谷啓介（訳）：アジャイルソフトウェア開発の奥義，ソフトバンククリエイティブ (2004).
12) CEDITI: Objectiver. http://www.objectiver.com/
13) CSK: VDM++ Toolbox. http://www.vdmtools.jp/en/
14) Sommerville, I.: *Software Engineering, 6th Edition*, Addison-Wesley (2000).
15) Gotel, O.C.Z. and Finkelstein, A.C.W.: An Analysis of the Requirements Traceability Problem, *Proc. 1st International Conference on Requirements Engineering*, pp.94–101 (1994).
16) Schneider, G. and Winters, J.P.: *Applying Use Cases*: *A Practical Guide*, Addison-Wesley (1998). 羽生田栄一（訳）：ユースケースの適用：実践ガイド，ピアソン・エデュケーション (2000).
17) Yu, E.: Modeling organizations for information systems requirements engineering, *Proc. 1st IEEE International Symposium on Requirements Engineering*, San Jose, pp.34–41, lEEE (1993).
18) 中島　震：ソフトウェア工学の道具としての形式手法，NIIテクニカル・レポート，国立情報学研究所 (2007).
19) IBM: Rational Rose. http://www-306.ibm.com/software/awdtools/developer/rose/
20) Dupuy, S., Ledru, Y. and Chabre-Peccoud, M.: An Overview of RoZ: A Tool for Integrating UML and Z Specifications, *CAiSE 2000*, LNCS 1789, pp.417–430, Springer (2000).
21) Snook, C.: Combining UML and B, *Proc. Forum on Specification & Design Languages*, Marseille (2002).
22) Bowen, J.P., Bogdanov, K., Clark, J.A., Harman, M., Hierons, R.M. and Krause, P.: FORTEST: Formal Methods and Testing, *Proc. Computer Software and Applications Conference 2002*, pp.91–101 (2002).
23) Beck, K.: *JUnit Pocket Guide*, O'Reilly (2004).
24) Nakagawa, H., Karube, T. and Honiden, S.: Analysis of Multi-Agent Systems based on KAOS Modeling, *Proc. IEEE/ACM International Conference of Software Engineering* (*ICSE 2006*), Shanghai, pp.926–929 (2006).
25) 中川博之，吉岡信和，本位田真一：IMPULSE：KAOSを利用したマルチエージェントシステムの分析モデル構築，情報処理学会論文誌，Vol.48, No.8, pp.2551–2565 (2007).
26) Nakagawa, H., Taguchi, K. and Honiden, S.: Formal Specification Generator for KAOS, *Proc. IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2007*), Atlanta, pp.531–532 (2007).

付　　録

A.1　VDM++仕様記述への変換アルゴリズム

```
/* エンティティの属性をセット */
for all ent in KAOS_entities do{
    ent が参照するエンティティを ent.attributes に追加;
    ent 内に属性が記述されていれば ent.attributes に追加;
}
```


© 2008 Information Processing Society of Japan

```
/* エンティティの判別 */
for all ent in KAOS_entities do{
   if(ent が agent からの"Control"線を持っている)
   then ent を agent.instanceList に追加;
   if(ent.attributes.size()==1 && ent.attributes(0).isSet())
   then ent を集合型とする;
   if(ent が agent.instanceList に含まれない && ent は集合型でない)
   then{
      if (ent.attributes.size()==0)
      then ent を VDM++_types に追加;
      else ent を VDM++_objects に追加;
   }
}

/* 共通定義クラスの生成 */
共通定義クラス用ファイルを open;
for all t in VDM++_types do {
   t の名前を記述;
   if(t.Def プロパティ!=null)
   then t.Def 値を VDM++の型として記述;
   else VDM++の型を記述するようコメントを追加;
   if(t.DomInvar プロパティ!=null)
   then t.DomInvar 値を不変条件として記述;
   else 不変条件があれば記述するようコメントを追加;
}
共通定義クラス用ファイルを close;

/* オブジェクトクラスの生成 */
for all obj in VDM++_objects do {
   obj 用オブジェクトクラスファイルを open;
   for all var in obj.attributes do {
      var のロール名を変数名，var の名前を型名として記述;
      if(var.DomInvar プロパティ!=null)
      then var.DomInvar 値を不変条件として記述;
      else 不変条件があれば記述するようコメントを追加;
   }
   obj 用のコンストラクタを記述;
   (引数として obj.attributes 内のすべての要素を記述)
   for all var in obj.attributes do
      var の Setter と Getter を記述;
   obj クラス用ファイルを close;
}

/* システムクラスの生成 */
for all ag in KAOS_agent do {
   ag 用システムクラスファイルを open;
   for all ins in ag.instanceList do {
      ins の名前を記述;
      if(ins.Def プロパティ!=null)
      then ins.Def 値を ins の型として記述;
      else ins の型を記述するようコメントを追加;
      if(ins.DomInvar プロパティ!=null)
      then ins.DomInvar 値を不変条件として記述;
      else ins の不変条件があれば記述するようコメントを追加;
   }
   for all op in ag.operations do {
      if(op.inputList.size=0 && op.outputList.size==0)
      then op の入出力型を記述するようコメントを追加;
      else op の入出力インタフェースを op.name と op.inputList,
           op.outputList から記述;
      op の内部処理を"is not yet specified"と記述;
      if(op.FormalDomPre プロパティ!=null)
      then op.FormalDomPre 値を op の事前条件として記述;
      else op の事前条件があれば記述するようコメントを追加;
      if(op.FormalDomPost プロパティ!=null)
      then op.FormalDomPost 値を op の事後条件として記述;
      else op の事後条件があれば記述するようコメントを追加;
   }
   ag 用システムクラスファイルを close;
}
```

(平成 19 年 10 月 18 日受付)
(平成 20 年 4 月 8 日採録)

© 2008 Information Processing Society of Japan

中川　博之（正会員）

1974 年生．1997 年大阪大学基礎工学部情報工学科卒業．同年鹿島建設（株）に入社．2007 年東京大学大学院情報理工学系研究科修士課程修了，2008 年同大学院博士課程中退．同年より電気通信大学助教，現在に至る．要求分析，形式手法，エージェント技術の研究に従事．電子情報通信学会会員．

田口　研治（正会員）

1956 年生．2001 年ウプサラ大学 Ph.D. in Computer Science 取得．CSK 総合研究所，（株）KRI，（株）CSK において，主に AI 関連の研究開発・コンサルティングに従事．1997 年九州大学助手，1999 年筑紫女学園大学助教授，2000 年ウプサラ大学講師，2002 年 ブラッドフォード大学講師，2005 年国立情報学研究所特任教授．形式手法，セキュリティ要求分析，ソフトウェア工学教育等の研究に従事．

本位田真一（フェロー）

1953 年生．1978 年早稲田大学大学院理工学研究科修士課程修了．（株）東芝を経て 2000 年より国立情報学研究所教授，2004 年より同研究所アーキテクチャ科学研究系研究主幹を併任，現在に至る．2008 年より同研究所先端ソフトウェア工学・国際研究センター長を併任，現在に至る．2001 年より東京大学大学院情報理工学系研究科教授を兼任，現在に至る．現在，早稲田大学客員教授，英国 UCL 客員教授を兼任．2005 年度パリ第 6 大学招聘教授．工学博士（早稲田大学）．1986 年度情報処理学会論文賞受賞．日本ソフトウェア科学会理事，情報処理学会理事を歴任．IEEE Computer Society Japan Chapter Chair，ACM 日本支部会計幹事，情報処理学会フェロー，日本ソフトウェア科学会編集委員長，日本学術会議連携会員．

© 2008 Information Processing Society of Japan