［論文］

エンティティの存在従属分析のためのドメイン特化言語
Domain Specific Language for Existence Dependency Analysis on Business Entities

井田 明男†, 金田 重郎†, 森本 悠介†
Akio Ida, Shigeo KANEDA, Yusuke MORIMOTO

†同志社大学大学院・理工学研究科
† Graduate School of Science and Engineering, Doshisha University

要旨

企業情報システムでは，システム相互の連携のために分散台帳技術や Web サービス API の採用が増すにつれて，業務で管理すべきエンティティをリソースとして可視化し，組織間で合意し，交換していくことの重要性が増してきている．具体的には，業務ドメインの概念モデルを適切に設計，維持しながら，それを元にデータベースを構築したり，構築した DB にアクセスするための API を提供していくことが開発者にとっての重要な責務となっている．概念モデルは従来，ER 図や UML クラス図などを用いて表記されてきた．これらの図は，人間にとっては読みやすいが，図そのものは機械可読ではなく，モデリングツールを用いたとしても，図から生成できるものは，RDB の Create Table 文，Java 等のプログラム言語の class 定義のスケルトンコードレベルに限られている上に，作図そのものも手間のかかる作業である．その上，エンティティの主キーと外部キーを人間が手作業で定義しなければならない．本来，主キーはオブジェクト id と同様，インスタンスを識別できる，業務の都合で決して変更されることのないものでなければならず，外部キーは参照先への関連の種類によって自動的に導かれるものである．このようなものを人間がいちいち定義し，モデルの可読性を低下させることはナンセンスである．その上，自然キーを誤って主キーに採用してしまうリスクも組み込まれる．そこで，本論文では，記述が容易で，人間，機械ともに可読な，存在従属分析結果の概念モデルを記述することに特化したドメイン固有言語（DSL）を提案する．提案 DSL は，試実装したコンパイラで処理することで，ドメインモデルのデータベース定義とそれらにアクセスするための API 仕様を得ることができる．また，提案 DSL では，エンティティの属性に履歴管理を指定することができ，履歴管理にはドキュメント指向のデータベースを併用する．本論文では，提案 DSL の構文と要素および意味論について説明し，7 つの例題を取上げてその記述性について確認した．

Abstract

In the enterprise information system, as the adoption of the distributed ledger technology and the Web service API increases for the mutual cooperation of the systems, it is important to visualize the entities to be managed in the business as resources, to agree on and exchange among the organizations. The need is getting stronger. Specifically, it is important for developers to properly design and maintain the conceptual model of the business domain, to build a database based on it, and to provide an API for accessing the DB that has been constructed It is a responsibility. Conceptual models have been conventionally written using ER diagrams, UML class diagrams, and the like. Though these figures are easy to read for human beings, the figures themselves are not machine readable, and even if modeling tools are used, those that can be generated from diagrams are RDB's Create Table statements, class definitions of program languages such as Java In addition to being limited to the skeleton code level. And the drawing itself is a laborious task. Also, a human must manually define the entity's primary key and foreign key. Originally, as with the object id, the primary key must be one that can identify the instance, never changed due to the circumstances of the business, the foreign key is automatically guided by the type of association to the reference destination . It is nonsense that humans define such things one by one and lower the readability and persistence of the model. Therefore, in this paper, we propose domain specific language (DSL) which is easy to describe, specialized in describing the conceptual model of existence dependent analysis result which is human and machine readable. Proposed DSL can be obtained by database implementation of domain model and API specification for accessing them by processing with trial implemented compiler. Also, in the proposed DSL, history management can be specified as an attribute of an entity, and a document-oriented database is used together for history management. In this paper, we explain the syntax and elements and semantics of the proposed DSL, and seven examples are discussed and its descriptiveness was confirmed.

［論文］
2018 年 9 月 16 日受付, 2018 年 12 月 12 日改訂, 2019 年 3 月 10 日受理
© 情報システム学会

1. はじめに

企業情報システム開発の分野では，システム間の相互連携のために RESTful な Web サービス技術[1]や分散台帳（ブロックチェーン）技術[2]の適用範囲が広がるにつれて，関係者間で管理すべきエンティティをリソースとして可視化し，合意し，交換していくことの重要性が増してきている．そのため，近年では図 1 に示すようなアーキテクチャの採用が進んでいる．従来は，サーバ上のアプリケーション（APP）から SQL[1]を発行して直接，基幹 DB[2]にアクセスしていたが，図 1 は，Web サービス API 層[3]を設け，サーバ APP からはこの層を介して間接的にアクセスする方式を示す．この方式により，基幹 DB 層[4]を Web サービス API 層によってカプセル化[5]することが可能となる[3].

このような状況に伴って，基幹 DB の設計はもちろんのこと，リソースの構造を反映した API 層の設計が重要課題となってきている．すなわち，業務で扱う重要な概念（エンティティ）とそれらの間の関係をモデル化した概念モデルを適切に設計，記述，維持しながら，それを元にデータベースを構築し，それらをサーバ・リソース[6]として，クライアントから容易にアクセスするための API を提供していくことが開発者にとって重要な責務となっている．よって，これからの開発に用いるツールは，概念モデル構築，データベース設計，API 設計といった一連の作業の整合性と合理化に資するものでなければならない.

概念モデルは従来，ER 図[7]や UML[8]クラス図などを用いて表記されてきた．筆者らは，概念モデル作成のガイドラインとしてエンティティの存在従属性に着目することを予てから提案してきた[4]が，ER 図ではモデル要素が僅かに不足し，UML クラス図にはモデル要素の種類こそ多いものの，そのままでは使いにくいといった問題があった.

そこで，本論文では，業務で扱うエンティティの存在従属性に着目した分析に用いるのに特化したドメイン固有言語（DSL）を策定し，関係データベース（RDB）をエンティティの永続化手段，Web サービス API 群をそれらへのアクセス手段とした開発への適用を提案する.

以下，第 2 章では提案手法の関連研究を概観する．第 3 章では，提案手法の詳細を述べる．第 4 章では，提案手法の実現性と記述性を確認する．第 5 章は，まとめである.

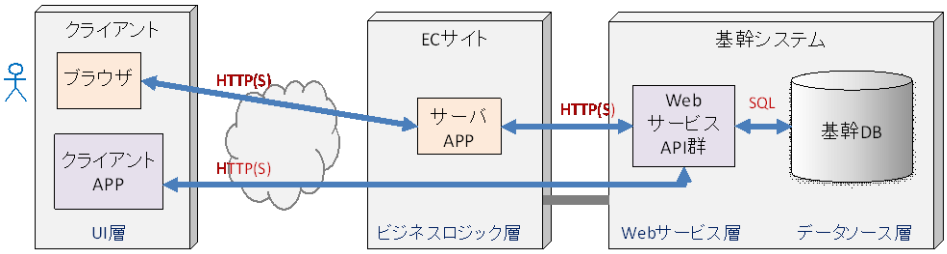

図 1　近年，採用が多いアーキテクチャ（Web サービス層にて基幹 DB をカプセル化）

2. 関連研究

2.1. エンティティの存在従属分析

エンティティの存在従属分析は，筆者らが予てから提案している，適切な概念モデルの構築を目的とした，エンティティのインスタンス間に見られる存在期間の依存関係および，それによって生じる制約に着目する手法である[4].

存在従属分析の概要は，まず，業務で捕捉，蓄積，管理すべき概念をエンティティクラス（以下，単にクラスまたはエンティティと記す場合がある）として切り出し，そのインスタンスと存在期間（管理期間）が等しい値をそのクラスの属性とする．次に，そのインスタンスは，前提なしで存在できるのか，あるいは，他のインスタンスの先立つ存在を前提として存在し得るのかを，業務要件に照らして検討していく．そして，インスタンス間に次の 4 つのタイプの依存関係を見いだす．すなわち，1) あるインスタンスとその属性値の関係（属性従属性），2) あるインスタンスとそのインスタンスの存在前提となる

---

1 関係データベース管理システム (RDBMS) において，データの操作や定義を行うためのデータベース言語
2 バリューチェーン上の主活動を支援するアプリケーション群（基幹システム）が使用するデータベース
3 アプリケーションからのデータベースへの Web 越しのアクセス提供するためのインターフェース
4 基幹システムのデータソース層
5 アクセス法だけを公開し，データの持ち方やアクセス法の実装は隠蔽すること
6 サーバ側に配置される，業務で捕捉・蓄積・管理すべき業務規則やデータ群
7 実体関連モデルを図で表記したもの
8 Object Management Group（OMG）がその仕様を管理する，記法の統一がはかられたモデリング言語

インスタンスの関係（存在従属性），3) あるインスタンスは別のインスタンスを参照するが，参照先のインスタンスが参照元のインスタンスの存在前提ではない関係（参照従属性），および，4) あるインスタンスは，別のインスタンスに対して共通の属性群と固有の属性群を持つ関係（汎化関係），これら 4 種類のインスタンス間の関係を，概念間の関係として一般化し，モデル化することで，概念モデルを構築する手法である．

なお，概念モデルは，対象領域に関わるさまざまなエンティティとそれらの関係を説明する実装独立なモデルであり，エンティティは，業務ドメインで捕捉，蓄積，管理すべき重要な概念であるため永続化が前提である．現状では，業務システムは RDB を用いてエンティティを永続化するケースが最も多く，SQL は十分に高水準な言語となっているため，概念モデルは実装独立とはいいつつも，SQL の DDL[9] 文に直結していて差し支えない，と筆者らは考えている．

存在従属分析を業務要件に即して忠実に行えば，正規化のプロセスなしで第 4 正規形以上の正規形レベルをもった概念モデルを得ることができるとされる[4]．そして，存在従属クラス図は存在従属分析の成果を表記するための有向グラフである．図 2 は物販業の受注管理ドメインについて存在従属分析を行った結果得られた，存在従属クラス図の例である．図 3a は，図 2 と等価で，図 2 の存在従属クラスが含意するとされる主キーと外部キーを文献[4]の記述に従って顕在化することで得られた ER 図である．ここで法人顧客の法人番号[10]，および個人顧客の個人番号[11]は自然キーであるため，顧客の主キーとして用いるのは適切でないことに注意されたい．また，図 3b は，注文エンティティの主キーを見直した ER 図である．3 章で述べる提案手法では，図 3b に表記された主キーの付与方式を採用する．注文エンティティの主キーを顧客 id と受注日時からなる複合主キー[12]から，単一属性の注文 id に変更する．顧客 id を非キー属性の外部キーに格下げする代わりに，注文から見た顧客のオプショナリティを排除することと，受注日時に非ナル制約を付すことで，図 3a のモデルと同等のデータ整合性の論理的強度を確保している．

なお，存在従属クラス図は UML クラス図の記法を流用しているため，静的な構造図に見えるが，実際はエンティティのインスタンスの存在期間に基づく依存関係を表現するので筆者らは動的モデルとしても位置づけている．日本語要求記述文からのエンティティの識別についての詳細は文献[5]を，存在従属という概念，および存在従属クラス図作成の詳細については，文献[4]を，それぞれ参照されたい．

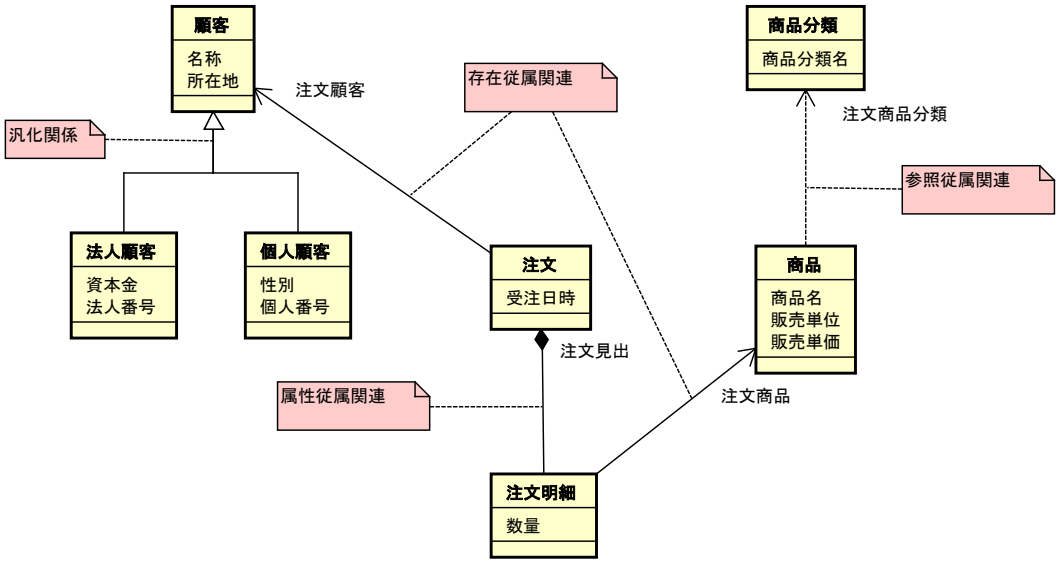

図 2 受注ドメインの存在従属クラス図の例

2.2. ドメインモデルとしての DSL について

DSL（domain-specific language）とは，特定のタスク向けに設計されたコンピュータ言語を意味する[6]．DSL は一種類の課題をうまく実行することに集中したものであるため，扱う問題に寄与しない言語要素は思い切って削ぎ落とすことが出来る点が DSL 採用の最大の利点であると考えられる．そのことによっ

[9] SQL のうち，データベースやテーブルの構造，およびビューの演算を定義する言語
[10] 法人向けのマイナンバーである 13 桁の数字
[11] 個人向けのマイナンバーである 12 桁の数字
[12] 複数の属性で構成される主キーであり，ユニーク制約と非ナル制約が含意される．

て，DSL を人間にとって理解しやすい簡潔さに保つことができる．

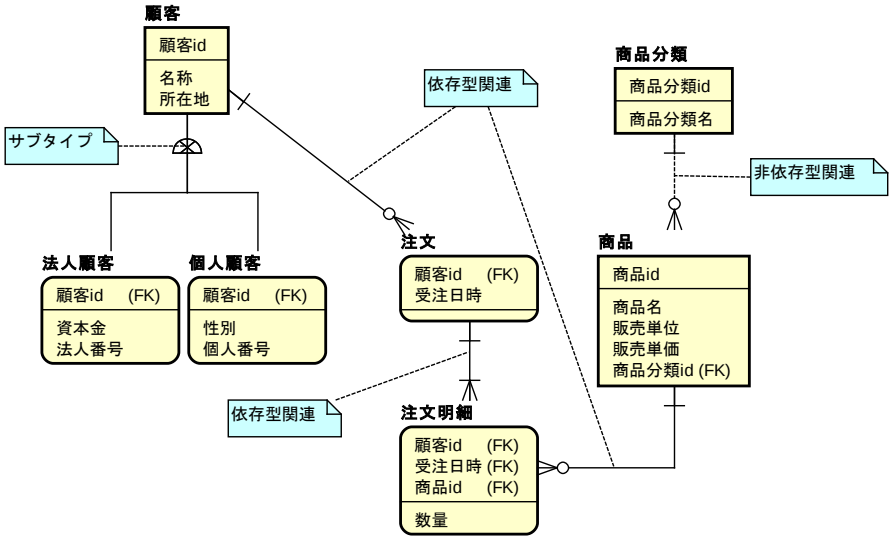

図 3a 図 2 を ER 図化したもの（文献[4]のガイドラインにて主キーを定義）

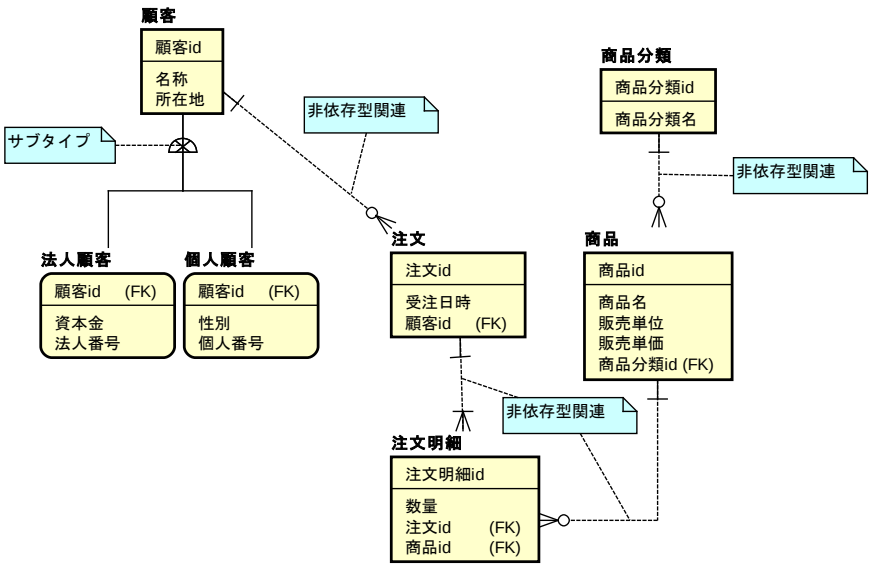

図 3b 図 3a の主キーを見直したもの（提案手法における主キーの付与方式）

従来，業務システムの核となるエンティティの構造を可視化するために，ER 図や UML のクラス図が用いられてきた．しかしながら，ER 図では，リレーションシップの種類が 3 種類（依存型，非依存型，サブタイプ）しかない上，あるエンティティから見た関連先のエンティティの役割を表記することができないため，現場で用いる場合には，注釈等で制約を補う必要がある．一方，クラス図では，関係の種類は 6 種類（依存関係，汎化関係，実現化関係，関連，集約，コンポジション）[7]と多いものの，業務システムの概念設計で使用するのは，関連，集約，コンポジション，汎化関係の 4 種類である．クラス図では，関連線は自由に引くことができ，関連の意味は関連名や関連端名によって柔軟に示せるものの，主キーや外部キーを明示的に表記するためには，限定子やステレオタイプを使って示す必要があるため，それらを愚直に行った場合，クラス図は複雑化し非常に見づらいものになってしまう．

存在従属分析では，使用するエンティティ間の関係は，属性従属関係，存在従属関係，参照関係，および汎化関係の 4 種類である．これは，ER 図のリレーションシップよりは 1 種類多く，UML クラス図のそれとは，種類数こそ 4 種類と等しいものの，それぞれの意味や含意する制約が微妙に異なるため，汎化関係を除いては，ストレートに対応づけることができない．たとえば，属性従属関連はコンポジション，存在従属関連は集約，参照従属関連は関連に近いと考えられるが，文献[4]に示されるように，そ

のまま 1 対 1 対応というわけではない．また，それらの表記法をサポートするモデリングツール群を開発ツールとしてみた場合は，モデルから DB 生成，DB にアクセスするための API 生成，および API のドキュメント生成へとシームレスにつなげられると便利であるが，現時点でそのような範囲を一貫してサポートするツールは存在しない，

3. 提案手法

そこで，筆者らは，存在従属分析の成果を書き表すための DSL を定義し，DLS4EDA（Domain Specific Language for Existence Dependency Analysis）と名付けた（以降，DSL4EDA と記す）．DSL4EDA は，次の事柄を達成すべくその仕様策定を行った．

a) 特殊なツールを用いることなく，テキストエディタだけでドメインの概念モデルが簡潔かつ機敏に記述できること．
b) UML クラス図の学習者が DSL4EDA に移行しやすいように，エンティティ間の関係を表す表記は，UML クラス図の図式表記に可能な限り似せること．
c) DSL の利点を活かして，DSL4EDA の要素は存在従属分析の結果を表記するために，必要最小限の構文とし，人間が読みやすく，習得のための学習コストを抑えること．
d) 存在従属分析結果のモデルを記述する際に，ER 図で足りない要素や，クラス図で表記しきれない要素を表記できるようにすること．
e) 表記されるエンティティやそれらの間の関係は，後続の機械処理によって，データベース定義や，定義されたデータベースにアクセスするための API 生成に直結できるものであること．特にデータベースへのアクセスを提供する API の実装は現在人気のある ORM 製品（Ruby の ActiveRecord[8]や PHP の Eloquent[9]など）を用いての実装から利用されるケースも考慮して，複合主キーは使わないこと．
f) モデル表記を人間にとって可読な簡素さに保つために，自明なことは，モデルに記述しないで済ませられるようにすること．表 1 は DSL4EDA において表記を省略する（あるいは省略できる）モデル要素の一覧である．

表 1　DSL4EDA では表記を省略する要素

表記を省略する要素	省略する理由
主キー	DSL4EDAでは，主キーは一切表記できない．その代わり人工キーであるidが自動的に定義される．これにより設計者が誤って自然キーを主キーとして採用してしまうリスクを回避する．なお，自然キーを非キー属性として定義することは妨げない
外部キー	依存関係のタイプによって自動的に決定されるため
多重度（カーディナリティおよびオプショナリティ）	依存関係のタイプによって自動的に決定されるため
属性の型が文字列型である場合の型指定	最も頻出する型であるため

3.1. DSL4EDA の仕様

DSL4EDA の文法を，図 4 に示す．図 4 では拡張バッカス・ナウア記法（BNF 記法）[10]を採用している． 図 4 では，ダブルクォート（”）で囲まれた文字列は，終端記号[13]であり，小なり記号（**<**）と大なり記号（**>**）で囲まれた文字列は，非終端記号[14]を表している．そして，縦棒記号（|）は，その前後にある終端記号や非終端記号のいずれかが選択されることを示す．また，プラス記号（**+**）は，直前の文字の 1 回以上の繰り返し，終端記号でないシャープ記号（#）以降，行末までは，文法の BNF 表記そのものの注釈である．

DSL4EDA では，たとえば，ドメイン名は，2 重の角括弧（**[[**および**]]**）で囲まれた識別名であり，エンティティ名は，角括弧（**[**および**]**）で囲まれた識別名である．そして，識別名は，特殊記号以外の文字の 1 回以上の繰り返しであることを表現している．例として，図 2 に示した存在従属クラス図を DSL4EDA の文法に則って表記した場合は，図 5 のようになる．

以下の節では，DSL4EDA の主だった文法と言語要素，および DDL への変換時の実装指定について，順に説明する．

[13] 具体的な文字や数字や記号で構成される，それ以上は変換されない記号
[14] 他の記号列と置換できるものとして定義されている記号

```
<特殊記号> ::= "," | "(" | ")" | "{" | "}" | "[" | "]" | "<" | ">" | "="
| "-" | "|" | ":" | "@N" | "@V" | "#" | "*" | "+"
<注釈> ::= "#" <文字>+ "¥n"
<識別名> ::= <特殊記号以外の文字>+
<ドメイン名> ::= "[[" <識別名> "]]"
<エンティティ名> ::= "[" <識別名> "]"
<属性名> ::= <識別名>
<役割名> ::= "(" <識別名> ")"
<属性型> ::= [ “string” ] | “number” | “dateTime”
<属性> ::= <属性名> [ ":" <属性型> ]["*"] # '*'は履歴管理したい属性に付与する
<属性並び> ::= "{" <属性> ["," <属性> ]+  "}"
# <属性並び反復> ::= "{" <属性並び> "}+"
<エンティティ型> ::= [ "@N" ] | "@V"
<エンティティ> ::= <エンティティ名> [ <属性並び> ]
<被役割> ::= <役割名>
<有向関連> ::= <被役割>"<>=" | <被役割>"<==" | <被役割>"1--" | <被役割>"<--"
| "=<>"<被役割> | "==>" <被役割> | "--1" <被役割> | "-->" <被役割>
<エンティティ関連> ::= <エンティティ> <有向関連> <エンティティ>
<特化関係> ::= <エンティティ> "<|-" <エンティティ>
<汎化関係> ::= <エンティティ> "-|>" <エンティティ>
```

図 4　DSL4EDA の構文規則

```
[[受注ドメイン]]
[顧客]{名称，所在地*}
[法人顧客]{資本金:number，法人番号}
[個人顧客]{性別，個人番号}
[注文]{受注日時:datetime}
[注文明細]{数量:number}
[商品]{商品名，販売単位，販売単価:number}
[商品分類]{商品分類名}
[法人顧客]-|>[顧客]
[個人顧客]-|>[顧客]
[顧客](注文顧客)<==[注文](注文見出)<>=[注文明細]==>(注文商品)[商品]-->(商品分類)[商品分類]
```

図 5 図 2 の存在従属グラフを DSL4EDA で表記したもの

3.1.1. 識別名

識別名は，DSL4EDA 自体で用いる特殊記号を含まない任意の文字列である．識別名は，エンティティ名，属性名，被役割名の一部または全部として用いられる．なお，特殊記号は半角文字を使用する．

3.1.2. ドメイン

DSL4EDA では，モデル化の対象領域に識別名を与えて，ドメインを定義できる．ドメイン名は，エンティティクラスの定義を格納するパッケージの名称，エンティティに対応した RDB のテーブル群を格納するデータベースの名称として使用することを想定している．ドメインは名前空間を形成し，次に述べるエンティティ名はドメイン内でユニークである必要がある．DSL4EDA では，あるドメイン定義から，次のドメイン定義が現れるか，あるいは，DSL ファイルの終端に出会うまでを，同一のドメインと解釈する．

3.1.3. エンティティ（Entity）と属性（Attribute）

【エンティティ】エンティティは，対象業務で捕捉，蓄積，管理すべき重要なデータ項目の塊である．何をエンティティとするかは，業務要件のみに依存する問題であるが，エンティティと属性の間の存在制約に着目することにより，かなりの精度でその塊を切り出すことは可能である[4]．DSL4EDA では，識別名を大括弧で囲むことにより，その識別名がエンティティの名称であることを表現する．たとえば，**[顧客]**，**[注文]**，**[商品]**の表記は，3 つのエンティティの存在と，それぞれのエンティティの識別名を表している．

【実装指定】1 つのエンティティは関係データベースの 1 つのテーブルにて実装する．エンティティ名が英語の場合，テーブル名はエンティティ名を複数形[15]にしたものに変換する．また，エンティティ名が日本語の場合は，エンティティ名の後ろに単純に英小文字半角の「s」を追加したものをテーブル名と

[15] 単複不規則変化の名詞については，https://stackoverflow.com/questions/18902608/generating-the-plural-form-of-a-noun に記載された算法に基づく処理を行う

する．

【エンティティが持つ属性群】エンティティは 1 つ以上の属性を持つ．属性は，エンティティの性質を規定するデータ項目である．ある値を，あるエンティティのインスタンスの属性値とするためには，両者の存在期間が等しくなければならない．エンティティが，どのような属性を持っているかを表記するために，エンティティ名の後ろに属性並びを表記することができる．たとえば，「**[顧客]{氏名，住所}**」という表記は，顧客はエンティティであり，属性並びにて，氏名，および住所と命名された属性を持つことを示している．

【エンティティの主キー】エンティティを識別するための主キー属性は id 方式[11][16]を採用する．RDB においては，主キー属性は識別性だけでなく，データの整合性を担保する役割を担う．たとえば，複合主キーは，複数の実在するエンティティの組合せに由来する制約を実装するのに簡潔かつ合理的な方法である．しかしながら，提案手法では，あえて id 方式を採用する．id 方式であるため，どのテーブルをとってもその主キーは, 複合主キーは出現せず, すべて単一属性の id である. id 方式を採用する理由は，1）id は連番等の意味のない値であるため，未来永劫，業務の都合で変更される心配がないため．2）すべてのエンティティの主キーは id である，と決めてしまうことによって表記を省略できるため．3）id 方式は Rails の ActiveRecord[8]や Laravel の Eloquent モデル[9]など，現在よく使われている ORM 製品から RDB を扱う際に親和性が高いため．4）複合主キーを持ったテーブルを参照するための外部キーは，やはり，複合属性になるため，属性数が多くなると，扱いや管理が煩雑になるため．といった理由による．id 方式であっても，非キー属性である外部キーの組み合わせに対してユニーク制約を設定することにより, 複合主キー兼外部キーを用いる場合と同等の整合性を実現することができる. 後述するように，DSL4EDA の実装指定では，外部キーまたは，その組合せに対して，非ナル制約やユニーク制約，連鎖削除等の設定を追加することによって，複合主キーが担う識別性以外のデータ整合性に関する機能を肩代わりさせる．

【属性並び】属性並びは，中括弧内に属性をコンマ区切りで並べたものである．モデルは，上から下へ読まれていくため，人間にとっての可読性のために，属性並びはエンティティが最初に登場する位置で表記することを推奨する．

【属性】属性は, その値がエンティティのあるインスタンスを特徴付け, その値の存在期間（管理期間）がエンティティのあるインスタンスのそれと一致するデータ項目である．属性として，属性名を表す識別名と属性の型を定義する．属性名の直後に，コロン「:」に続けて，属性の型を指定する．属性の型は，「文字列（string）」,「数値（number）」,「日時（dateTime）」が指定できる[17]．ビジネスドメインでは，文字列型が使用されるケースが圧倒的に多いため．属性の型が文字列型である場合に限り，その表記を省略してよい．

【属性値の履歴管理】属性の直後にアスタリスク（*）を付与することで，その属性の値については履歴管理すべきことを指定できる．

履歴管理の実装指定は，図 6 のように，RDB と MongoDB[12]に代表されるドキュメント指向データベース（以下，DDB と記す）を併用して実装するのが現時点における最適解であると考えている．なぜならば，RDB はあらかじめ定義されたスキーマに従って，格納されるデータの整合性を CUD[18]時に厳密にチェックするため，実行速度は多少犠牲なるものの，データの一貫性は RDBMS の機能によって担保される．一方，DDB は，データの一貫性の維持は犠牲にしても，高速なアクセスが必要で，スキーマを柔軟に成長させていくような用途に向いている．両者の特性を組み合わせることで，データの一貫性を犠牲にすることなく，柔軟かつ高速なデータ管理が実現できると考えられるからである．

図 6 は顧客エンティティの顧客住所が履歴管理指定された場合の実装イメージである．DSL4EDA の処理結果が実行される時点で，RDB 側に顧客テーブルが作成されるとともに，DDB 側には，顧客住所履歴を追記していくためのコレクションが作成される．顧客のインスタンスが新規登録されるタイミングで，RDB 側には，当該顧客の新規レコードが登録されるとともに，DDB 側には，当該顧客の住所履歴を記録していくためのドキュメントが挿入され，そのドキュメント id が，RDB 側の住所履歴属性に値として登録される．そして，顧客の住所が更新されるタイミングでは，RDB 側の顧客住所の値を最新

[16] id 方式は，すべてのテーブルの主キーを意味のない連番にするやり方であり，文献[10]では「id リクワイアド（とりあえず id）」という名称のアンチパターン（不適切な解決策）として紹介されている．

[17] 現時点では 3 種類であるが，業務システムの概念モデルではこの 3 種類の属性型で足りそうである．

[18] レコードの新規追加（Create）,レコードの更新（Update）,およびレコードの削除（Delete）

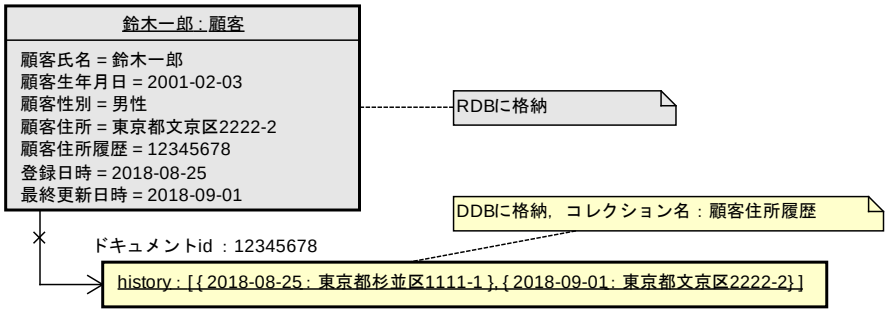

図 6　RDB と DDB を併用した属性値の履歴管理のイメージ

の値で更新するとともに, DDB 側の当該顧客の住所履歴に関するドキュメントには新しい住所とその更新日時のペアが追記される. こうすることで, RDB 側の顧客テーブルには, 従来通りアクセスできる上, 顧客の過去の住所が必要になった場合には，アクセス時の引数に時点を追加指定することにより，指定された時点における住所の値を取得することができる.

3.1.4. 有向関連（Directed Association）

DSL4EDA で扱う 2 つのエンティティ間の関連は，属性従属関連，存在従属関連，参照従属関連の 3 種類であり，それらはすべて，依存元から依存先に向けての向きを持った有向関連である．依存元からみた依存先への多重度は，「1」または「0..1」に限定されるが，関連のタイプによってそれぞれ独特の制約を含意する．なお，図 7 に示されるように，3 つの関連種のうち，参照従属関連には参照先の多重度に応じて，さらに 2 つのバリエーションがあり，それぞれ右向きと左向きの表記を許すため，結果的に 8 つのバリエーションが生まれる．

```
<有向関連> ::= <被役割>"<>=" | <被役割>"<==" | <被役割>"1--" | <被役割>"<--"
             | "=<>"<被役割> | "==>"<被役割> | "--1"<被役割> | "-->"<被役割>
```

図 7　有向関連の表記のバリエーション

依存先には，依存元から見た役割を，「被役割（responsibility)」として表記する必要がある．被役割は，識別名を小括弧で囲ったもので，UML クラス図における関連端名（ロール名）に相当する要素である．たとえば，「注文」エンティティから見た「顧客」エンティティの被役割を「**(注文顧客)**」として定義する．被役割はあとあと外部キーの名前にしたり，仮想表としてリソース化し，API 経由で完結したリソースとして検索，取得できるようにする際のリソース名としての使用に有用である．

ちなみに，関連端名（ロール名）のモデル要素は，UML クラス図には存在するが，ER 図では，現在主流である IE[19]表記，IDEF1X[20]表記，それらいずれの表記においてもサポートされていない．

3.1.4.1. 属性従属関連(Attributive Dependency)

【表記】属性従属関連は，「**<>=**」 または 「**=<>**」の記号で表記する．等号が置かれる側に依存元，その反対側に依存先のエンティティを記述する．なお，依存先のエンティティには被役割を付記するが，属性従属関連の場合に限り，その名称が依存先のエンティティ名と同じものを指定することができる．そして，その場合に限り，被役割の表記は省略してよいが，この場合は，被役割名として，依存元エンティティ名に依存先エンティティ名を結合した名称が採用される．

【適用】この関係は，この記号で接続された 2 つのエンティティのインスタンスが有するライフサイクルが等しい場合に限り，適用することができる．

【適用例と実装指定】属性従属関連の適用例としては，「注文」エンティティと「注文明細」エンティティの関係がある．これらは，本来，概念としては 1 つであるが，通常，1 回の注文において，注文明細は 1 件以上反復されるため，「注文明細」は「注文」から分離され，本体のエンティティ「注文」に向けて属性従属関係が定義されることになる．この例の DSL4EDA による表記は，「**[注文](注文見出) <>=[注文明細]**」のように書ける．

属性従属関連を RDB にて実装する場合は，依存側のエンティティに，非ナル制約を付与された外部キーが定義され，その外部キーには連鎖削除が指定されるようにする．

---

19 James Martin が提唱した情報システム設計開発方法論 Information Engineering (IE)における ER 図の表記法

20 米国標準技術研究所（NIST： National Institute of Standards and Technology）が規格化した ER 図の表記法

「[注文] (注文見出) <>=[注文明細]」の例から，生成すべき DDL 文は図 8 のようになる．注文明細

```
create table 注文s (
    id              integer primary key,
);
create table 注文明細s (
    id              integer primary key auto_increment,
    注文見出id      integer not null
    foreign key(注文見出id) references  注文s(id) on delete cascade
);
create view 注文見出s as
select 注文s.id , 注文明細s.id
From   注文s  inner  join  注文明細s on 注文s.id = 注文明細s.id;
```

図 8 属性従属関連の実装例

テーブル（注文明細 s）から注文テーブルに向けての非ナル制約付きの外部キーが生成され，さらに，その外部キーには連鎖削除が設定される．

なお，後で API から注文そのものを 1 つの完結したリソースとして参照できるようにすべく，個々の注文明細と注文とを結合した仮想表（view）の DDL を，前述の被役割名（「**注文見出**」）を用いて生成する．

3.1.4.2. 存在従属関連（Existence Dependency）

【表記】存在従属関連は，「**<==**」 または，「**==>**」 の記号を用いて表記する．等号が置かれる側に依存元，反対側に依存先のエンティティを記述する．

【適用】この関連は，依存元エンティティのインスタンスが存在できるためには，依存先のエンティティのインスタンスの先立つ存在が前提であり，かつ，依存元エンティティのインスタンスの存在期間を通じて，依存先のエンティティのインスタンスは消去も挿げ替えも行わない場合に限り適用することができる．結果として，依存元からみた依存先の多重度は厳密に 1 であり，依存先エンティティのインスタンス（これは依存元のインスタンスにとっての存在前提であるため）を挿げ替えたり消去したりすることは許されない上，依存元のインスタンスが 1 つでも存在している間は，依存先のエンティティの当該（依存されている）インスタンスは削除できないという制約が実現できる．

【適用例と実装指定】存在従属関連の適用例としては，「納品」と「請求」の関係がある．「納品」のインスタンスが存在しなければ「請求」のインスタンスを作れないとする業務要件は頻繁に見受けられる．この場合，**[納品](請求対象)<==[請求]**のように表記する．

存在従属関連を RDB にて実装する場合は，依存側のエンティティに，非ナル制約を付与された外部キーが定義される．この例の場合，図 9 のような DDL 文を出力する．

```
create table 納品s (
    id    integer primary key auto_increment,
);
create table 請求s (
    id    integer primary key auto_increment,
    請求対象id      integer not null,
    foreign key(請求対象id) references 納品s(id)
);
create  view  請求対象s   as
    select 納品s.id, 請求s.id
    from 納品s inner jopin 請求s on 納品s.id = 請求s.id;
```

図 9　存在従属関連の実装例（その 1）

存在従属関連に関しては，業務ドメインでよく現れる別の例を見よう．たとえば，「在庫」は「倉庫」と「商品」に存在従属するエンティティである．提案 DSL の表記例は，「**[倉庫](在庫倉庫)<==[在庫]==>(在庫商品)[商品]**」となる．実在する倉庫のインスタンスと商品のインスタンスの組み合わせによって，在庫の諸属性が一意に特定できる．この場合，図 10 のような DDL 文を出力する．

```
create table 倉庫s (
    id    integer primary key auto_increment,
);
create table 商品s (
    id    integer primary key auto_increment,
);
create table 在庫s (
      id          integer primary key auto_increment,
      在庫倉庫id  integer    not null,
      在庫商品id  integer    not null,
      unique(在庫倉庫id, 在庫商品id),
      foreign key(在庫倉庫id) references  倉庫s(id),
      foreign key(在庫商品id) references  商品s(id),

);
create view 在庫倉庫在庫商品s as …
```

図 10 存在従属関連の実装例（その 2）

3.1.4.3. 参照従属関連（Referencing Dependency）

【表記】参照従属関連は，「**<--**」または「**-->**」の記号を用いて表記する．マイナス記号が置かれる側に依存元（参照する側），反対側に依存先（参照される側）のエンティティを記述する．

【適用】この関連は，依存元エンティティのインスタンスと依存先エンティティのインスタンスのライフサイクルは独立しており，依存元エンティティのインスタンスの存在期間を通じて，依存先のエンティティのインスタンスを消去したり挿げ替えたりしたい場合に限り適用することができる．

【適用例と実装指定】参照従属関連の例として，「野球選手」と「球団」の関係を採り上げよう．この関係は，ある野球選手インスタンスの存在期間を通じて，所属球団は不在であったり，挿げ替えられたりされる．DSL4EDA では，「**[野球選手]--> (所属球団) [球団]**」のように表記できる．

参照従属関連を RDB で実装する場合は，依存側のエンティティに，参照先のエンティティのインスタンスを特定するための外部キーが定義されるのが適切である（図 11）．なお，「野球選手は常時どこか１つの球団に必ず所属していなければならない」といったルールがある場合には，ある選手から見て，所属球団の不在は許されない．そのような場合は「**<--**」の代わりに「**1--**」を，「**-->**」の代わりに「**--1**」を使用することができる．両者の違いは，その RDB 実装において依存元の依存先に向けての外部キーに非ナル制約が設定されるか，されないかの違いである（図 12）．

```
create table 球団s (
    id    integer primary key auto_increment,
);
create table 野球選手s (
    id    integer primary key auto_increment,
    所属球団id      integer,
    foreign key(所属球団id) references 球団s(id)
);
```

図 11　参照従属関連の実装例（参照先の多重度が 0..1 の場合）

```
create table 球団s (
    id    integer primary key auto_increment,
);
create table 野球選手s (
    id    integer primary key auto_increment,
    所属球団id      integer not null,
    foreign key(所属球団id) references 球団s(id)
);
```

図 12　参照従属関連の実装例（参照先の多重度が 1 の場合）

【備考】上記の例は，所属期間，年俸など，所属にまつわる諸属性を管理したい場合などは，参照従属関連としてではなく，所属そのものをエンティティとして切り出し，野球選手と球団にそれぞれ存在従属関連を定義して，「**[野球選手](所属野球選手)<==[所属]{開始日，終了日，年俸}==>(所属球団)[球団]**」のようにモデル化する必要がある．

3.1.5. 汎化関係（Generalization）および特化関係（Specialization）

【表記】汎化関係および特化関係は「**<|--**」 および 「**--|>**」の記号を用いて表記する．マイナス記号が置かれる側に特化されたエンティティ（サブクラス）を，反対側に汎化されたエンティティ（スーパークラス）を書く．

【適用】この関係は，「サブクラスはスーパークラスの一種である」，という文章が成り立つ場合に限って適用可能である．

【用例と実装指定】たとえば，「**[個人顧客]-|>[顧客]**」，または「**[顧客]<|-[個人顧客]**」という表記は，「顧

客」がスーパークラスで「個人顧客」がサブクラスであることを表現する．なお，DSL4EDA では，

```
[法人顧客] -|> [顧客]  または  [顧客] <|- [法人顧客]
[個人顧客] -|> [顧客]  または  [顧客] <|- [個人顧客]
```

図 13　DSL4EDA における複数のサブクラスが共通のスーパークラスを持つ場合の記法

```
create table 顧客s (
    id    integer primary key auto_increment,
    # 顧客の属性群
);
create table 法人顧客s (
    id    integer   primary key,
    # 法人顧客特有の属性群
    foreign key(id) references  顧客s(id)
);
create table 個人顧客s (
    id    integer   primary key,
    # 個人顧客特有の属性群
    foreign key(id) references  顧客s(id)
);
```

図 14　汎化関係の実装例

表 2　DSL4EDA で表記するエンティティ間の依存関係

関係の種類	意味	依存先（参照先）エンティティのインスタンスの挿げ替え	提案DSLにおける表記	依存先への外部キーへの制約
属性従属関連	依存元のエンティティは依存先のエンティティの属性であるが，構造をもった繰り返し属性項目であるため，別のエンティティとして分離されている	不可	[依存先]<>=[依存元] または [依存元]=<>[依存先]	参照整合性制約，非ナル制約，ユニーク制約，連鎖削除を設定する
存在従属関連	依存元エンティティのインスタンスは依存先エンティティのインスタンスの先立つ存在を前提条件として存在可能である	不可	[依存先]<==[依存元] または [依存元]==>[依存先]	参照整合性制約，非ナル制約，依存先が2つある場合は，その組合せについてユニーク制約を設定する
参照従属関連	参照元のエンティティのインスタンスと参照先のエンティティのインスタンスの存在期間は独立しており，参照先インスタンスの不在を許す	可	[依存先]<---[依存元] または [依存元]--->[依存先]	参照整合性制約，非ナル制約を設定する
	参照元のエンティティのインスタンスと参照先のエンティティのインスタンスの存在期間は独立しており，参照先インスタンスの不在を許さない	可	[依存先]1---[依存元] または [依存元]---1[依存先]	参照整合性制約を設定する
汎化・特化関係	「[サブクラス]は[スーパークラス]である」という文が成り立つ．スーパークラスの主キーはサブクラスでも共用され，それがスーパークラスのインスタンスへの外部キーを兼ねている	不可	[スーパークラス]<|-[サブクラス] または [サブクラス]-|>[スーパークラス]	参照整合性制約，非ナル制約，ユニーク制約，連鎖削除を設定する

1 つのスーパークラスに対して，複数のサブクラスが存在する場合は，図 13 のように複数行に分けて記述する必要がある．

汎化および特化の関係を RDB にて実装する場合には，スーパークラスの主キーをサブクラス側の主キー兼外部キーとして共有する形にする．この例では，図 14 のようにする．

以上，DSL4EDA の主だった文法と言語要素，および DDL への変換時の実装指定について，順に説明してきたが，表 2 にエンティティ間の依存関係についてまとめておく．

3.2. PlantUML との比較

ここで，DSL4EDA の類似ツールについても確認しておきたい．UML の世界では，PlantUML[13]というオープンソースのツールが 2009 年頃から存在する．PlantUML もプレーンテキストでモデルを表記するため，表記されたモデルの見た目は DSL4EDA のそれとよく似ている．しかしながら，前者がプレーンテキスト言語からグラフィカルベースの UML ダイアグラムを作成できるツールであるのに対して，後者は，データベース定義およびそれにアクセスための API の仕様を生成できるツールであり，立脚しているモデル要素の意味論も前者は UML[7]のそれであるのに対して，後者は，インスタンスと属性値および，それらの存在期間の時間的前後関係に由来する制約であるため，両者は，似て非なるものである．

3.3. DSL4EDA コンパイラの試実装

本提案の目的は，単に概念モデルをコンパクトに表記できることではない．DSL にて記述された概念モデルを元に，データベースを構築した上で，クライアントから，それらに容易にアクセスできるようなインターフェースの提供までを一貫して行うことで，開発を支援することである．そのためには，DSL4EDL で表記されたモデルを読み込んで，基幹 DB（RDB）のためのスキーマ定義，履歴管理 DB（DDB）のためのコレクション定義，さらに，生成される DB 群（以下，リソースと記す）にアクセスするための API 仕様を出力するようなコンパイラが提供されてしかるべきであろう．

そこで，筆者らは，DSL4EDA コンパイラを試実装した．試実装は，DSL4EDA の実現可能性の確認も兼ねている．いかに使いやすい DSL の文法を定義できたとしても，それが実装できなければ，DSL の価値がなくなるためである．図 15 にコンパイラの入力と出力を示す．なお，コンパイラが生成するコード群は，第 1 章の図 1 で示したアーキテクチャの下で使用することを想定しており，Web サービスの設計・実装者，サーバ APP の設計・実装者，およびクライアント APP の設計・実装者のそれぞれに価値が提供できることを意図している．

Web サービスの実装者は，コンパイラが出力した API 仕様を参照しながら，公開する Web サービスのメソッド部を実装していくことができる．そして，その時点では，RDB 用のスキーマ定義（SQL の DDL 文），および履歴管理用 DDB のコレクション定義が提供されている．一方，サーバ APP の設計・実装者，およびクライアント APP の設計・実装者は，コンパイラが出力した API 仕様文書を参照しながら，アプリケーションのビジネスロジック部分を実装していけばよいことになる．ビジネスロジック部分の実装時に必要なデータの操作は，Web サービスに委譲されるため，ビジネスロジックの実装に専念することが可能となる．

DSL4EDA コンパイラの試実装は Ruby 言語で行い，パーサ部の実装要素技術として，Treetop[21][14]を用いた．本論文では DSL4EDA の文法を拡張 BNF で記載しているが，試実装では，これを PEG[22]形式[15]に直した後，Treetop に読込ませることで，パーサを得ることにした．そのため，トークンの切り出しやトークンタイプの解釈といった処理を自前で実装する必要がないため，少ないステップ数で実装できた．

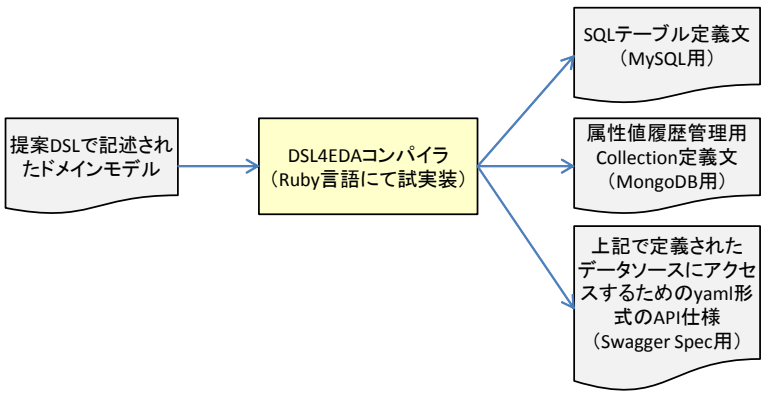

図 15　DSL4EDA コンパイラの入力と出力

3.3.1. DSL から SQL テーブル定義文の出力

ここで，コンパイラが行うべき処理は，DSL に記述されたそれぞれのエンティティの定義を，エンティティと 1 対 1 で対応する，RDB テーブルのスキーマ定義（CREATE TABLE 文），および，有向関連が表記された DSL の各行に対応する仮想表のスキーマ定義（CREATE VIEW 文）を生成して出力することである．

CREATE TABLE 文の生成では，テーブルごとに，属性並びで指定されたカラム定義に加えて，id という名称の主キー属性を追加[23]し，属性の型は，3.1.3 節の記述に従って定義する．また，当該エンティティに依存先のエンティティがある場合には，3.1.4 節および 3.1.5 節にて紹介した，依存関係の種類に応じた実装指定に基づき，外部キー制約及びカラム制約を追加する．さらに，アスタリスク（*）が付与されて履歴管理が指定された属性については，後で述べる DDB のコレクションにおける履歴管理のド

[21] PEG 形式で記述された構文規則の定義を読み込んで，パーサを出力するツール

[22] Parsing Expression Grammar の略で，形式言語をその言語に含まれる文字列を認識するための一連の規則を使って表したものを指す

[23] 他にも，レコードが新規登録された時点を保持するために created_at，最後に更新された日時を保持するために last_modified_at という属性が自動的に追加されるが詳細は割愛する．

キュメント id を保持するための属性を数値型の属性として追加しておく．

CREATE VIEW 文の生成では，DSL に定義された依存関係を読み込む都度，依存元テーブルの外部キーと依存先テーブル主キーの一致を内部結合の条件とした SELECT 文を，仮想表の中味として生成する．その際，射影する属性は，依存元のテーブルの属性群と依存先のテーブルの属性群の和集合をものである．

なお，テーブル，仮想表を問わず，その名称は，3.1.3 節の記述に従って，DSL に表記されたエンティティ名または被役割名を複数形化（識別名が英語の場合は，Ruby が提供する pluralize 操作を使用し，日本語の場合は，半角英小文字の's'をエンティティの識別名に追加）したものを使用する．

3.3.2. DSL から属性値履歴管理用コレクション定義の出力

ここで，コンパイラが行うべき処理は，DSL 上で履歴管理が指定された属性毎に DDB 用のコレクションを作成することである．試実装では DDB 製品として，MongoDB を採用することにした．コレクション名は，エンティティ名と履歴管理する属性名を連結した名称である．3.1.3 節で述べた通り，コンパイラは，履歴管理が必要な属性を検出した時点で，所定の DDB（デフォルトの名称は，「<ドメイン名>historiesdb」）に対して，db.createCollection 文を発行するスクリプトを出力して，コレクションの生成に備える．

3.3.3. DSL からデータソースにアクセスするための API 仕様の出力

ここで，コンパイラが行うべき処理は，生成したテーブル毎に，そのテーブルに対してクライアントとなるアプリケーションからネットワーク越しにアクセスできるように CRUD 操作[24]を行うための WebAPI 仕様（API 仕様）を生成することと，生成した仮想表についても同様に，個々の仮想表に対してクライアントから Read 操作を行えるように API 仕様を生成することである．そして，テーブルと仮想表をまとめてリソースと呼ぶ．

API 仕様の要素は，HTTP 操作，URL で指定される操作対象のリソース名，リクエスト時のパラメータ，およびレスポンスの内容で構成される．DSL4EDA コンパイラでは，URL の名称として，リソースがテーブルの場合は，エンティティ名を，仮想表の場合は被役割名を用いている．たとえば，「get /顧客」にて，顧客の一覧が取得でき，「get /注文顧客」とすると，（API 呼び出し時点で）注文を持っている顧客の一覧を取得できるといった API 仕様を生成することができる．DSL4EDA コンパイラは，テーブルおよび仮想表に対して，それぞれにアクセスするための API 仕様を生成するが，表 3 にそれらの基本形を示す．

表 3　DSL4EDA が生成する API 基本形

操作対象	定義する操作	HTTP操作	URLパタンの基本形	リクエスト・ボディの内容	レスポンス・ボディの内容
テーブル	レコードの新規登録	POST	/{エンティティ名}	新規登録するインスタンスの属性と属性値のオブジェクト表現	新規登録されたインスタンスの属性と属性値のオブジェクト表現
	既存のレコードの更新	PUT	/{エンティティ名}/{id}	更新するインスタンスの属性と属性値のオブジェクト表現	更新後のインスタンスの属性と属性値のオブジェクト表現
	すべてのレコードの読出し	GET	/{エンティティ名/	なし	読み出されたインスタンス群の属性と属性値のオブジェクト表現の配列
	idで指定されたレコードの読出し	GET	/{エンティティ名}/{id}	なし	読み出されたインスタンスの属性と属性値のオブジェクト表現
	idで指定されたレコードの指定年月日時点のおける属性値での読出し	GET	/{エンティティ名}/{id}/At/{年月日}	なし	読み出されたインスタンスの属性と属性値のオブジェクト表現
	属性値によるレコードの検索	GET	/{エンティティ名}/findBy{属性名}/{属性値}	なし	読み出されたインスタンス群の属性と属性値のオブジェクト表現の配列
	idで指定されたレコードの削除	DELETE	/{エンティティ名}/id	なし	削除されたインスタンスの属性と属性値のオブジェクト表現
仮想表	すべてのレコードの読出し	GET	/{仮想表名}/	なし	読み出されたインスタンス群の属性と属性値のオブジェクト表現の配列
	属性値によるレコードの検索	GET	/{仮想表名}/findBy{属性名}/{属性値}	なし	読み出されたインスタンス群の属性と属性値のオブジェクト表現の配列

[24] 新規作成（Create），読み出し（Read），更新（Update），削除（Delete）についての一連の操作

図 16　DSL が出力した API 仕様を Swagger Editor で開いた様子

当然，API 呼び出し時に，時点を示すパラメータが追加指定された場合には，DDB 側もアクセスして，該当する時点の属性値を返却する API を用意しなければならない．具体的には，文献[1]の第 4 章，リソース指向アーキテクチャの統一インターフェースに関するガイドラインに則って URL パターンを定義し，その呼び出し法のドキュメントをマークアップ言語にて生成する．試実装では，ドキュメント作成ツールとして Swagger[16]を採用したため，WebAPI の仕様は YAML 形式で書き出される．試実装では，筆者らは，本論文に先だって，存在従属グラフから RESTful Web サービスを生成する提案を行っていた[17]が，その実装を DSL4EDA のモデルを入力とするように改良したものを使用した．図 16 は DSL が出力した API 仕様を Swagger Editor で開いた様子である．

4. 評価実験

DSL4EDA の記述性を確認するために，表 4 に示す 7 つのモデリング題材[25]に対して，存在従属分析を行い，その分析結果を DSL4EDA の文法で問題なく記述できることを確認した．本節では，その中から，DSL4EDA で扱うほとんどの関係種が登場する，題材番号 5 番の題材を採り上げて説明する．図 17 は，表 4 の題材番号 5 番の概念モデルに反映すべき要件のリストである．そして，図 18 は，図 17 の要件に対して存在従属分析を行った結果を，DSL4EDA で表記したモデルである．DSL4EDA で意図したとおり，簡潔に題材の概念モデルを記述することができた．次の 4.1 節では，図 17 から図 18 を導出した過程を説明する．

表 4 DSL4EDA の記述性を確認するために用いた題材の一覧

題材番号	題材の概要
1	学生が科目を履修し，教員が科目を担当し．科目が参考書を指定するドメイン
2	日々のフライト運航便へパイロットと客室乗務員を割り当てるドメイン[18]
3	バントはメンバーで構成され，バンドは，パートを要請し，メンバーはパートを担当するドメイン
4	提供科目に制約がある学級の実施教室に制約がある科目に対して，登壇日に制約がある講師を割り当てるドメイン
5	町内リレーの走行区間に走者を割り当て，出来事を記録するドメイン
6	酒屋の在庫管理問題のドメイン[19]
7	顧客に単品花で構成される花束を指定日に組み立てて出荷するドメイン[20]

[25] 文献[18]，[19]，[20]のようなデータベースやモデリングの教科書に記載された題材および，筆者らが大学の講義や勉強会で使用している題材

```
■ 走者には，名称がある：具体例：Aさん，Bさん，Cさん，など
■ 地点には，名称と所在地がある：具体例：
  W地点，東京都港区1丁目，X地点，東京都世田谷区2丁目，など
■ ２つの地点の間を区間と定義した場合，区間は2つの地点なしでは，存在し得ない
■ 区間には，出発地点から到着地点への方向性がある（逆走できない），
  よって区間からみた２つの地点を，それぞれ出発地点，到着地点として区別する必要がある
■ 区間が決まれば，距離が定まる，ただし，直線距離ではなく地点を結ぶ道の距離とする
■ 区間は，地点を介して隣接する，すなわち，スタート地点からゴール地点に向けて，
  前の区間と次の区間が高々１つ存在する
■ 走者には必ず1つの区間が割り当てられる（そして走者はその区間を走行する）
■ 走者は，区間の始点と終点において必ずリレーのトリガ（競技トリガ）を実施する，
  すなわち，つぎのいずれかである：
  ・走者はスタートして，受け持ち区間を走りきり，
    次の走者にバトンタッチするか（この場合，前の走者は存在しない）
  ・走者は前の走者からバトンタッチを受けて，受け持ち区間を走りきり，
    次の走者にバトンタッチするか（この場合，前の走者と次の走者がいずれも存在する）
  ・走者は前の走者からバトンタッチを受けて，受け持ち区間を走りきり，
    ゴールするか（この場合，次の走者は存在しない）
■ イベントには，それが起こった地点（発生地点）と発生時刻が帰属する．
  そして，それらはリレーの事象として，逐一記録したい
```

図 17 評価実験のためのモデリング例題

```
[[リレー競技ドメイン]]
[地点]{名称，所在地}(出発地点) <== [区間]{距離} ==> (到着地点)[地点]
[走者]{氏名} --1 (担当区間)[区間]
[競技トリガ]（走行開始トリガ）1-- [走者] --1（走行終了トリガ）[競技トリガ]
[バトンタッチ] -|> [競技トリガ]{発生時刻:dateTime} ==>(発生地点)[地点]
[スタート]     -|> [競技トリガ]
[ゴール]       -|> [競技トリガ]
[走者](前走者)<==[バトンタッチ]==>(後走者)[走者]
```

図 18　提案 DSL による分析結果の表記

4.1. 導出過程

4.1.1. エンティティとその属性を識別する

例題の問題記述から，業務で捕捉，蓄積，管理すべき概念をエンティティとして切り出し，そのインスタンスと存在期間（管理期間）が等しい値をそのクラスの属性として識別する．この例題の場合，「走者」，「地点」，「区間」，「スタート」，「バトンタッチ」，「ゴール」，そして「競技トリガ」がエンティティである．そして，「走者」の「氏名」，「地点」の「名称」と「所在地」，「区間」の「距離」，そして，「スタート」，「バトンタッチ」，「ゴール」の「発生時刻」，および「競技トリガ」の「発生時刻」，「発生地点」が，それぞれのエンティティのインスタンスと存在期間が等しいため，それぞれの属性として識別する．なお，属性値として識別された値の中で，反復的に記述される値がある場合には，その値を，別のエンティティのインスタンスとして分離し，後続のステップで属性従属関連を定義することになる．なお，この例題の場合は，反復的な繰り返し属性は現れなかった．

4.1.2. エンティティ間の依存関係を識別する

次に，そのインスタンスは，前提なしで存在できるのか，あるいは，他のインスタンスの先立つ存在を前提として存在し得るのかなどを，業務要件に照らして検討し，2 つのインスタンス間の依存関係を識別していく．識別すべき依存関係は次の 1)から 4)のいずれかであり，識別作業は順不同で行う．

1） 属性従属関連を識別する

２つのエンティティのインスタンス間に，例外なくライフサイクルの一致が認められる場合は，それを属性従属関連として識別することができる．エンティティの属性は 4.1.1 節にて識別済みであるが，繰り返し的な属性を識別した結果，別エンティティのインスタンスとして分離していた場合は，属性従属関連を定義し，分離したエンティティから元のエンティティに向けて属性従属関連を定義する．

2） 存在従属関連を識別する

２つのエンティティのインスタンス間に，依存元エンティティのインスタンスが存在できるためには，依存先のエンティティのインスタンスの先立つ存在が前提であり，かつ，依存元エンティティのインスタンスの存在期間を通じて，依存先のエンティティのインスタンスは消去も挿げ替えも行わないような関係が認められる場合は，それを存在従属関連として識別することができる．この例題の場合，「区間」は 2 つの「地点」に存在従属する，と認識できたため，「区間」から「地点」に向けて．存在従属関連を定義した．その際，リレーの区間は，出発地点から到着地点に向

けての方向性があるため，それぞれの地点について，「出発地点」，「到着地点」と呼ぶ被役割を定義した．また，「スタート」，「バトンタッチ」，「ゴール」についても，「地点」に存在従属すると認識される．さらに，「バトンタッチ」は，前走者と後走者の存在が前提として発生する出来事であるため，「バトンタッチ」を，「前走者」と「後走者」という被役割を定義した「走者」の間に定義し，「バトンタッチ」からそれぞれの走者に向けて存在従属関連を定義した．

3） 参照従属関連を識別する

2 つのエンティティのインスタンス間に，依存元エンティティのインスタンスと依存先エンティティのインスタンスのライフサイクルは独立しており，依存元エンティティのインスタンスの存在期間を通じて，依存先のエンティティのインスタンスを消去したり挿げ替えたりしたい関係が認められる場合には，それを参照従属関連として識別することができる．この例題の場合，走者には必ず 1 つの区間が割り当てられるため，「走者」から「区間」に向けて，多重度が必ず 1 の参照従属関連を定義する．また，すべての走者は，自分の受け持ち走行区間の始点と終点において，必ず 1 回ずつのリレーのトリガ（競技トリガ），これには，「スタート」，「バトンタッチ」，あるいは「ゴール」というバリエーションがある，を経験するため，「走者」から「走行開始トリガ」および「走行終了トリガ」という被役割を定義した「競技トリガ」に向けて，多重度が必ず 1 の参照従属関連を定義した．

4） 汎化関係を識別する

2 つのエンティティクラス間に，「サブクラスはスーパークラスの一種である」，という文章が成り立つ場合は，それを汎化関係として識別することができる．この例題の場合，『「スタート」は「競技トリガ」の一種である』，『「バトンタッチ」は「競技トリガ」の一種である』，そして，『「ゴール」は「競技トリガ」の一種である』，という文章がいずれも成り立つため，「スタート」から「競技トリガ」へ向けて，「バトンタッチ」から「競技トリガ」へ向けて，そして「ゴール」から「競技トリガ」へ向けて，それぞれ汎化関係を定義するとともに，それまでサブクラス側の属性および関連であった発生時刻，および発生地点をスーパークラス側に移動した．

概ね，以上のような分析を行うことによって，要件についての記述を，DSL4EDA に変換することができる．評価実験では，表 4 の題材番号 1〜4，6，7 の題材についても，同様な分析を行い DSL4EDA にて概念モデルを簡潔に記述できることを確認した．

5. まとめ

本論文では，存在従属分析によって得られた概念モデルの記述に特化した DSL を提案し，その文法とモデル要素の意味，およびモデルを RDB の DDL スキーマに変換する際の実装指定を説明した．エンティティの主キーを id と定めて，エンティティ間の関係から自動的に外部キーが設定されるようにする，などで，表記を大幅に簡素化しながらも，後続の開発作業で必要となる，リソース定義へと結びつけていくことに成功している．

そして，データベースの教科書等に記載されている題材を評価実験のためのモデリングの題材として，その記述性について確認した．結果，すべての題材について，存在従属分析結果を DSL4EDA で表記でき，同時に試実装のコンパイラで意図したとおりの出力結果が得られることを確認した．これにより，DSL4EDA の文法は，パーサにとっても無理のない構文であることを確認できた．記述性と可読性と機械処理可能性をすべて満たすのは一般に難しいことであるが，DSL4EDA はそれらを評価できる水準で満たしていると思われる．

概念モデルを DSL4EDA の様式で記述しておくことにより，その後の開発作業を機械化しやすい．DSL4EDA はテキストエディタとコンパイラさえあれば，存在従属分析手法に基づいたドメインモデルを手軽に作成・編集し，データベース定義とアクセスメソッドを瞬時に手に入れて，設計者間や開発環境間で共有できることが最大の売りである．

DSL4EDA コンパイラは試作段階ではあるが，この実現には，最新のソフトウェア工学の成果に依るところが大きい．Ruby などのプログラミング言語や treetop[14]や Swagger[16]といった優れたツールが存在しなかったならば，このような DSL を試実装することさえできなかったと思われる．このような優れた技術開発に日々取り組んでおられるコミュニティの方々に深く感謝したい．

今回の評価実験の題材は教材の域を出ない小さなものであったため，今後は，DSL4EDA をさまざまな分野の業務における概念設計に適用しながら，その文法を改良していくとともに，コンパイラの機能や品質を向上させてゆきたいと考えている．

参考文献

[1] L. Richardson, S. Ruby, “RESTful Web サービス”, オライリージャパン, 2007 年 9 月.
[2] 清水響子, “「ブロックチェーン」の基本を理解してみる”, https://it.impressbm.co.jp/articles/-/14647, (2017 年 6 月 20 日確認).
[3] 渡辺幸三, “設計者の発言-マイクロサービスで周辺アプリと連係する”, http://watanabek.cocolog-nifty.com/blog/2015/12/post-7a2e.html, 2015-12-14.
[4] 井田明男, 金田重郎, 熊谷聡志, 藤本明莉, “存在従属性に着目した論理要件ロバストなドメインモデルの作成ードメインクラス図をユビキタス言語として用いるためにー”, 情報処理学会論文誌, Vol.56, No.5, pp.1340-1350, 2015 年 5 月.
[5] 金田重郎, 井田明男, 酒井孝真, 熊谷聡志, “日本語仕様文からの概念モデリングガイドラインー行為文と関数従属性に基づくクラス図の作成ー”, 電子情報通信学会論文誌 D, Vol.J98-D, No.7, pp.1068-1082, 2015 年 7 月.
[6] Martin Fowler, “ドメイン特化言語 パターンで学ぶ DSL のベストプラクティス 46 項目”, ピアソン桐原, 2012 年 5 月.
[7] OMG, “OMG Unified Modeling Language Superstructure, Version 2.4”, https://www.omg.org/spec/UML/2.4/About-UML/, 2010-11-14
[8] “Active Record - Object-relational mapping in Rails”, https://github.com/rails/rails/tree/master/activerecord, (2018 年 9 月 8 日確認).
[9] “Eloquent: Relationships”, https://laravel.com/docs/5.7/eloquent-relationships, (2018 年 9 月 8 日確認).
[10] “Extended Backus-Naur form”, https://en.wikipedia.org/wiki/Extended_Backus_Naur_form, (2018 年 9 月 8 日確認).
[11] Bill Karwin, “SQL アンチパターン”, オライリージャパン, 2013 年 1 月.
[12] “MongoDB”, https://www.mongodb.com/, (2018 年 9 月 8 日確認).
[13] “PlantUML”, http://plantuml.com/, (2018 年 11 月 28 日確認).
[14] “Treetop - A Parsing Expression Grammar (PEG) Parser generator DSL for Ruby”, https://rubygems.org/gems/treetop/versions/1.6.8, (2018 年 9 月 8 日確認).
[15] Bryan Ford, “Parsing Expression Grammars:A Recognition - Based Syntactic Foundation”, Massachusetts Institute of Technology, 2004 年 1 月.
[16] OpenAPI Initiative, “Swagger The Best APIs are Built with Swagger Tools”, https://swagger.io/, (2018 年 9 月 8 日確認).
[17] 井田明男, 金田重郎, 森本悠介, “存在従属グラフから RESTful Web サービスの生成”, 第 12 回 情報システム学会 全国大会・研究発表大会, P014, 2016 年 11 月.
[18] 増永良文, “リレーショナルデータベース入門[第 3 版]”, サイエンス社, 2017 年 2 月.
[19] 児玉公信, “UML モデリングの本質”, 日経 BP 社, 2011 年 5 月.
[20] IT 勉強宴会, “花束問題 v2”, https://groups.google.com/forum/#!topic/benkyoenkai/W3qnr2OGLC4, (2018 年 9 月 8 日確認).

著者略歴

井田 明男（いだ あきお）

1984 年同志社大学文学部文化学科（心理学専攻）卒業．銀行員，大手メーカ系 SE を経て 1989 年 4 月，株式会社オージス総研入社．業務系および技術系のシステム開発やオブジェクト指向と UML を用いた開発のトレーニングやコンサルティングに従事．2012 年に独立．現在，有限会社井田代表取締役，同志社大学理工学部講師．要求デリング，構造モデリング，コード自動生成の研究に従事．

金田 重郎（かねだ しげお）

1974 年京都大学工学部電気第二学科卒業，1976 年 3 月京都大学大学院工学研究科電子工学専攻修士課程修了．同 4 月，日本電信電話公社・武蔵野電気通信研究所入所．大型汎用電子計算機の実用化，並びに，誤り検出訂正符号の研究に従事．1997 年 4 月同志社大学大学院総合政策科学研究科教授・同工学部教授．現在は，同理工学研究科・情報工学専攻教授．要求分析・センシング応用技術の研究に従事．工博（京都大学），技術士（情報処理部門）．

森本 悠介（もりもと ゆうすけ）

2017 年 3 月同志社大学理工学部インテリジェント情報工学科卒業. 2017 年 4 月同志社大学理工学研究科情報工学専攻博士前期課程入学．要求モデリング手法，コード自動生成の研究に従事．