Effort Used to Create Domain-Specific Modeling Languages

Juha-Pekka Tolvanen
MetaCase
Jyväskylä, Finland
jpt@metacase.com

Steven Kelly
MetaCase
Jyväskylä, Finland
stevek@metacase.com

概要

ドメイン固有のモデリング言語とジェネレータは、システムとソフトウェア開発の生産性と品質を大幅に向上させることが証明されています。ただ、これらの利点は通常、言語、ジェネレータ、および関連ツールを作るための初期投資の規模を説明せずに報告されています。

そこで MetaEdit+ツールを使用して、特定のドメインのための完全なモデリングソリューションを開発する取り組みに焦点を当てながら、2 つの異なる方法で合計 10 の事例にわたって必要とされる投資について比較します。まず、ケーススタディを行って、2 つの現実的なサイズのドメイン固有のモデリングソリューションを実装する開発努力に関する詳細なデータを紹介します。次に、さまざまな企業から公開されている 8 つの事例をレビューして、同じツールを使用した業界の経験からデータを取得し、それらを私たちのケーススタディの結果と比較します。ケーススタディと業界事例の両方から、このツールでは、ドメイン固有のモデリング言語環境を作成するために必要な投資はわずかであることが示されました。その範囲は 3〜15 人日、平均 10 人日です。

CCS CONCEPTS
• **Software and its engineering** → *Domain specific languages*; *Model-driven software engineering*;

KEYWORDS
Domain-specific language, modeling, code generation, development cost

ACM Reference Format:
Juha-Pekka Tolvanen and Steven Kelly. 2018. Effort Used to Create Domain-Specific Modeling Languages. In ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark. ACM, New York, NY, USA, 10 pages.
https://doi.org/10.1145/3239372.3239410

1　はじめに

ドメイン固有言語（DSL）とドメイン固有モデリング（DSM）は、自動車、家電、信号処理、電気通信、太陽光発電システム、軍事などのさまざまな業界でソフトウェア開発の生産性と品質を大幅に向上させることが示されています[4, 15, 27, 32]。これらの改善を可能にするのは、次の 2 つの特徴です。まず、DSL は、特定の問題空間のコンセプトとルールを直接使用して解決策を指定する言語によって、プログラミングの抽象化レベルを上げることができます。第二に、ジェネレータが高いレベルの仕様から完全に機能的なコードを生成します[11, 27]。抽象化のレベルが自動化と伴って上がるたびに、その改善は著しいものであることが、歴史的に示されています。

通常、DSM または DSL の利点は、元の投資の規模を詳述することなく報告されます。モデリング言語、ジェネレータ機能、およびツールサポートを作るのに要した人数と所要時間などです。この調査では、10 の事例にわたって、完全な DSM ソリューションを作成するための取り組みを比較することによって、この不足に対処します。

また 2 つの補完的な方法で言語作成の取り組みを分析することによって、投資の規模を調査します。ケーススタディを自分で行うことと、報告された業界事例を分析することです。私たちは、現実の状況で作成され使用される言語に焦点を合わせます。これらの言語実装は完成されて、使用する準備ができています。ケーススタディ研究方法を使用して、我々は 2 つのケースで詳細に実装努力に関するデータを取ることに加えて、開発工数を報告した 8 つの産業界の言語開発プロジェクトの資料を調べます。すべてのケースで比較した結果、適切なツールを使用した場合、特に達成された成果と比べて、開発作業はわずかであると言えます。

この調査の研究方法と主題を説明した後、我々のケーススタディと業界事例を報告し、それらの取り組みを比較します。それから、他の公表された言語開発努力の比較を調べて、我々の結果をその一連の証拠に統合することによって結論を下します。

2　言語作成作業の調査

DSL を作成するのは難しく[15]、特に各種ツールのサポートの作成も含まれる場合[19]、多くの時間とリソースが必要になると言われています。残念ながら、モデリングソリューションの作成に関する研究の大多数は、その取り組みについてはあまり開示していません。また DSL の作成を調査することも、比較に見合った一般化されたデータを入手するのがむつかしく、困難です。まず、ドメイン間に明らかな違いがあるため、言語作成プロジェクトを比較するのは困難です。第二に、言語開発者の経験に違いがあります。 第三に、言語開発に適用されるツールや技術は、必要な努力に影響を与えます。この調査では、同じツール MetaEdit+[9, 17]での言語作成の取り組みに焦点を当てることによって、さまざまなツールの影響を除外します。

2.1　調査の主題：DSM ソリューション

「言語を作成する」とは、言語ユーザーに提供できる完全なモデリングソリューションを作成することを意味します。通常、完全な DSM ソリューションを実現するには、以下の部分を規定する必要があります。

メタモデル：言語の作成は、ほとんどの場合、言語のコンセプトの識別から始まります。コンセプトそのものは、言語を形成するための特性と関係性で具体化されます。結果の言語は、メタモデルまたはツールの文法に形式化されます。

制約：言語のコンセプトに加えて、モデルが従うべき多くのルール（規則）や制約もあります。それらを定義することはモデリング時のエラーの防止と早期発見を助け、仕様作成作業を加速し、そしてモデルをジェネレータ、チェッカー、シミュレーション等に適用可能にします。これらのルールは、通常、メタモデルの一部であるか、制約を表現する言語で個別に定義されます。

表記は仕様を作成、編集、および読み取るための表現を与えます。通常、シンボルはモデリング言語の主なコンセプトごとに定義されています。さらに、誤りや不完全さを示し、読み易さを提供するための表記要素を持つこともあります。多くの場合、表記は図表による表現ですが、マトリックスや表など他の表現パラダイムに基づくこともあります。この研究では、ドメイン固有モデリングに焦点を当て、テキストによる表現とプログラミング言語の拡張については、構文指向のテキストエディタとともに除外します。

ジェネレータはモデルを読み込み、それらをコードまたは他の出力に変換します。ジェネレータを構築することは、モデル要素がコードまたは他の出力にどのようにマッピングされるかを定義することです。ジェネレータの構築作業には、最適化、パターンとコーディングスタイルの使用、既存のライブラリと従来のコードとの統合など、結果として得られるコードが理想的であることの確認も含まれます。

ツーリングとツールチェーン：使用されているツールによっては、言語のメタモデル、制約、および表記を定義することに加えて、モデリングツール機能を実装するための追加情報またはコードを提供する必要があるかもしれません。ジェネレータもツーリングに依存し、他の一連のツールと統合する必要があります。コンパイラ、ビルドプロセス、要求管理、バージョン管理、テストなどです。

さらに、言語作成者は、ドメイン固有モデリング言語環境のサポートを提供するために必要なもの（トレーニング教材、チュートリアルの例、ガイドなど）に加えて、補助教材を作成するかもしれません。これらは他のタスクやテクノロジーのものと似ており、セットはどちらかといえばオープンエンドなので、ここではついでに見るだけにします。文献レビュー中に、上記のすべての部分の実装作業を詳述した出版物は見つかりませんでしたが、ケーススタディではそれを可能にします。

2.2 研究法

言語開発の取り組みについてより包括的な見解を得るために、2 つの研究方法を適用します。ケーススタディと、業界レポートのレビューです。

2.2.1 ケーススタディ 事例研究の方法で、2 つの DSM 作成プロジェクトに関する詳細なデータを収集します。1 つは組み込みデバイス用のアプリケーションの作成を扱い、もう 1 つは ArchiMate[28]と呼ばれるエンタープライズ系のアーキテクチャ設計言語を実装します。これらは現実的なサイズであり、またその実装は制限なしに利用できることを理由に選択しました。これらの言語は、その目的、そして既存の仕様が提示されている方法、特に制約やジェネレータが異なります。

すべての研究と同様に、ケーススタディは、科学的知識を得ることに加えて、それに続く研究アプローチの限界を克服または最小化するために計画されなければなりません。私たちの研究目的は、DSM ソリューション全体のさまざまな部分を作成するための開発努力に関する詳細な情報を抽出することです。

ケーススタディでは、言語の作成は典型的な反復的アプローチ[11、33]に従い、小さな部分で言語を定義し、実際の例で言語を用いることで評価します。これらの言語開発は、他のほとんどの業界事例（セクション 4 を参照）と同様に 1 人のエンジニアにより実装されて、言語ユーザーに使用されることでテストされました。言語エンジニアは、どちらのケースもドメインの詳細には精通していませんでしたが、DSM ソリューションを実装しながら、それらドメインについて学びました。ただ DSM ソリューションを定義して使用するツールについては既に熟知していました。そして言語のテストは、言語ユーザーにリファレンスとなるアプリケーションで結果を確認し、利用可能な場合は認証文書を使用するなどを依頼することによって行われました。

どちらの場合も、モデリングのサポートは完全に実装され、メタモデル、表記、およびセマンティクスをサポートツールとともにカバーしています。これら両方の DSM ソリューションは、統合されたヘルプおよびサンプルモデルと共に使用できるようになっています。

言語サポートの実装中に、主に 2 つの方法でデータを収集しました。労働時間の手動記録の保存と、ツールによって提供されるログデータの使用です。後者は自動的に収集されるためより正確ですが、DSM ソリューションのすべての部分に関するデータを提供するわけではありません。MetaEdit+は、特定の言語コンセプトが作成されたとき（秒単位のタイムスタンプ）、言語定義作業が開始されたとき、および作業を保存することによって停止されたとき（トランザクションの時間）にログ情報を提供できます。ジェネレータが作成または変更されたときのタイムスタンプもツール内で検査できます。

自動的に収集されたログに加えて、言語エンジニアはモデルとメタモデルをカバーする変更をバージョン管理することができます。自動的に収集されたバージョンと手動で作成されたバージョンの両方を使用して、開発期間に関する詳細な情報が得られました。MetaEdit+は表記に関連するログ情報を自動的に収集しないため、表記の作成に関連する作業は手動で保存された記録に基づいています。

私たちが行ったのと同じ要素に従って言語実装の取り組みに関するデータを収集することによって、ケーススタディを再確認することができます。将来の研究の範囲はまた、例えば他の言語を実装することによって、または異なるツールを使用することによって拡張することができます。

2.2.2 業界事例のレビュー　ケーススタディに加えて、私たちはさまざまな企業やドメインからの、公に利用可能な事例をレビューしました。これにより、ケーススタディの結果と業界事例を比較し、言語作成の取り組みのより具体的な実態を得ることができました。ただし言語エンジニアとしての著者の影響を避けるために、著者が直接関わった業界事例は除外しました。

企業は通常、一般的なレベルでも投資や努力について報告しないため、業界の経験の調査は予想よりも困難でした。[6]の 50 の事例のうちほんの数例だけです。企業は投資に関するデータを共有したくない、または必然的にデータ収集に時間をかけたくないでしょう。もしデータが提供される場合でも、言語作成に用いたツールの各種作業を識別できないように表現されます。例えば、Ericsson Modeling Days の講演[1]では、次の事だけが説明されています。Eclipse modeling tools をベースにした特定のモデリングツールの導入に 5 年以上を費やし、モデリングツールの作成に 35 人のスタッフが携わりました[3]。

我々は、開発努力に関する詳細が報告された、企業内で作成された特定の DSM ソリューションを説明した論文とプレゼンテーションに焦点を当てました。レポートは、ドメイン固有モデリング（domain-specific modeling）またはドメイン固有およびモデル（domain-specific, model）のキーワードを使用して検索エンジンのクエリ結果から選択され、業界事例であることが明確なレポートに絞りました。したがって、例えば、産業界のカンファレンスの記録に掲載されていても、研究者だけが行った事例は除外されました。言語の作成を説明する学術出版物は、言語自体やその使用法、または関連するツールに焦点を当てる傾向がありますが、実際の開発努力に関する情報はめったに含まれません。そして、特定の側面を研究することに焦点が当てられているため、完全ですぐに使える DSM ソリューションを提供するのではなく、必ずしも完全に定義されているわけではない（たとえばメタモデルのみ）ことも良くあることです。

この調査ではツールによる影響を抑制するために、以前の研究において高い評価を受けた[10, 12, 13]、1 つのツールに焦点を合わせました。そして MetaEdit+ベースの DSM ソリューションに限定した文献が選択されました。

3　言語開発の取り組みの事例２つ

言語開発者は、ドメインおよび/または言語に関連するすべての文書にアクセスできました。最初のケースでは、モノのインターネット（IoT）のデバイス製品用に DSM ソリューションを開発しましたが、そのドメインのコンセプトを識別することは、デバイスのセンサーと機能といった明白なセットがあるため、非常に簡単でした。これらは全てデバイスのドキュメント（マニュアル）に詳述されています。また、これらのコンセプトを 1 つの言語にまとめるのは比較的簡単でした。これにはドメイン固有の状態モデルのアプローチが適切と思われました。表記、モデル検査のためのルール、およびジェネレータを特定して開発する必要がありました。コードジェネレータは利用可能なリファレンスアプリケーションでテストされました。

ArchiMate メタモデルの実装は、The Open Group による既存の仕様に従ったため、ほとんどすべての要件が利用可能なケースといえます。したがって、それは既知の言語が定義されている場合の努力を比較するために行われた他の研究と似ています（例えば[13、21]）。ArchiMate の実装は、参照モデルを作成し、利用可能な認証文書を使用してテストされました。しかしながら、我々は費用がかかることを懸念して認証プロセスをたどりませんでした、また驚くことに認証[30]は詳細には実装をカバーしていませんでした。例えば ArchiMate 言語に関連する多数のルールが省略されています。

3.1　IoT デバイス

我々が実装した最初の言語は、IoT デバイス用のアプリケーション開発を対象としています。ロジックをカバーして、センサーデータを読んで、そして外界と通信するアプリケーションコードを開発者が手動で実装するのではなく、2 つのモデリング言語を統合して、JSON ジェネレータを加えた DSM ソリューションを作成しました。そのため、アプリケーション開発者は、センサーやサービスなどの IoT デバイスのコンセプトを使用してアプリケーションを設計し、そのモデルから完全なコードを生成して、それを IoT デバイス（https://www.thingsee.com/）で実行することができます。

言語のコアは、デバイスのサービスとルールで拡張されたステートマシンをベースにしています。センサー（動き、圧力、輝度、GPS、温度など）は、電池残量や充電状態の確認などハウスキーピング機能とともにイベントやデータを提供します。ステートマシンのアクション部分は、デバイスが外界とやり取りできるさまざまな通信方法（クラウドへのメッセージ送信や携帯電話への SMS 送信など）を網羅しています。図 1 は、実装された DSM ソリューションを使用して開発された小さな例を示しています。温度センサーと湿度センサーを使用するサウナアプリケーションです。温度が 60℃を超えると、アプリケーションは湿度をチェックし、指定された携帯電話番号に情報を SMS 送信します。この例は 10 を超えるセンサーの 2 つを利用して SMS 送信のみが使用されている小さなものです。図 1 では、モデルを作成し、それらをチェックし、ジェネレータを実行するためのモデリング環境を示しています。

統合したモデリング言語の一つは単一のアプリケーションの実装に焦点を当てていますが、二番目の IoT 言語は、開発者が複数のアプリケーションまたはモジュールを統合することを可能にします。これは、図 1 のようにスケーラビリティに取り組み、さまざまなモデルを管理するのに役立ちます。

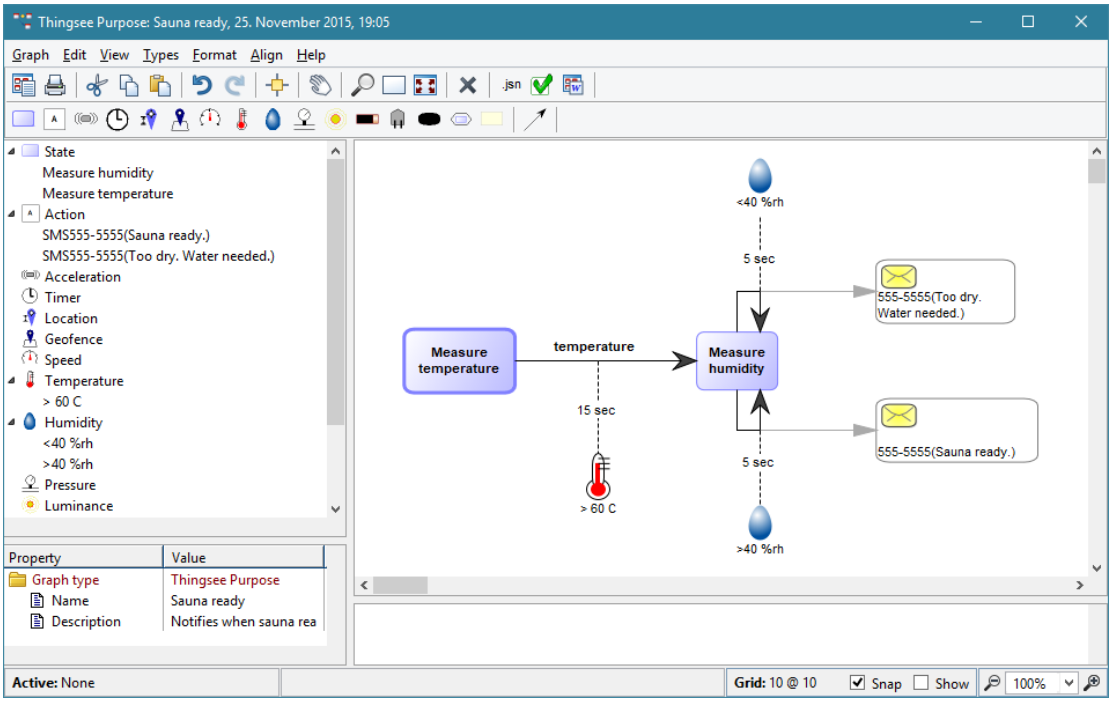

図 1　使用されている IoT デバイス言語

3.1.1 実装された DSM ソリューションの一部　完全に定義されたメタモデルは 74 個の要素を持ちます。参考までに、これは同じメタモデリング言語（MetaEdit+）内の UML 2.5 [20]のメタモデルのサイズの約 3 分の 1 です。さらに、メタモデルに 39 個の制約が定義されました。これには、リレーションシップでオブジェクト間をどのように接続できるかについてや、そのような接続に参加できる候補数、および大規模アプリケーションの設計を処理するためにダイアグラムを編成する方法が含まれます。最も一般的な種類の制約は、入力したプロパティ値がドメイン内で正しいことを確認するための正規表現です。湿度、圧力などの適正な値、または値の設定を必須にすること。これらの制約により、モデルに入力されたデータが正しいことが保証され、そのままコード生成に

使用できます。メタモデルと制約を開発する中、メタモデルのコンセプトは必要に応じて説明で注釈されたので、モデリングツールは言語ユーザーに統合されたヘルプシステムを提供できるようになっています。

表記には 54 個のシンボル要素を定義しました。表記要素の多くは、モデルに入力された設計データに応じて異なるシンボル要素を示す条件付きでした。さらに、9 つの小さなシンボルがアイコンとして使用されるように定義されました。これらのアイコンは必須ではありません － MetaEdit+はメインシンボルに基づいて自動的にアイコンを提供します － しかし、要素が選択されるエディタのツールバーから言語のコンセプトが識別されやすくなります。これらのアイコンは、さまざまなビューやブラウザでコンセプトを区別するためにも使用されます。

ジェネレータは、2 つの目的で定義されています。1 つはメタモデルで表現されていないルールや制約をチェックするためのもの - 例えば、開始状態の欠落や他のデザインに接続されていない要素などモデル（設計）上の不完全さをチェックします。もう一つはデバイスにアップロードされる JSON コードを生成するためです。チェック機能については、モデリング時の制約が既に効いているため 1 つのジェネレータモジュール（MERL（MetaEdit+ Reporting Language）で 21 行）で十分でした。JSON ジェネレータは 28 のジェネレータモジュール（556 行）で構成されています。

これらの定義に基づいて、MetaEdit+はモデリング、モデルの管理、共同編集、モデルの妥当性のチェック、そしてコードの生成のためのツール機能を提供します。その完全な言語とジェネレータは、https://www.metacase.com/download から入手できる MetaEdit+の評価版の中に「IoT」サンプルプロジェクトとして含まれています。

この言語は完全なものですが、組み込まれたガイダンス機能を参考にして、拡張することもできます。例えばセンサーがバッテリーを消費しているかどうかを知らせるための仕様変更などです。ただし安全のために、デバイス製造業者の保証より高い温度を測定することのないようにチェックもされます。これは、DSM ソリューションでドメイン固有の知識を直接提供する典型的な例です。

参考資料：MetaEdit+のメタモデリング機能 https://www.metacase.com/ja/mwb/

3.1.2　DSM の実装努力　すぐに使える IoT モデリングをサポートする言語開発の総実装作業量は 3.8 人日でした（1 日 8 時間）。DSM ソリューション全体のさまざまな部分に費やされる実装作業を図 2 に示します。メタモデルの実装とその制約およびチェック機能は、わずか 1 人日以内に行われました。正確には、自動的に収集されたログに基づいて、メタモデルは 5.8 時間で定義されました。MetaEdit+の制約設定機能で直接表現可能な 39 のルールの定義は 1 時間でした。モデルの完全性と正当性をチェックするジェネレータの実装にさらに 1 時間かかりました。

オブジェクトと、それらを接続するためのリレーションシップやロールのシンボル表記には 5 時間かかりました。

JSON コードを生成するジェネレータの実装には 17 時間かかりました。デバイス製造元によって提供されたサンプルアプリケーションのいくつかをモデル化し、生成されたコードを同じアプリケーション用に提供された参照コードと比較することによって、ジェネレータの正しさをチェックしました。その後、テスト駆動アプローチに従って、ジェネレータは多数の小規模なリファレンス

アプリケーションに対して詳細にテストされました。[31]で詳述されているようなテストモデルの作成はジェネレータ開発作業に含まれていません。

モデリング時に必要になるエディタ、ブラウザ、ヘルプシステムなどは MetaEdit+が提供するので、言語のツールサポートを実装するための作業はほとんどゼロでした。必要な唯一のツーリング関連部分は、モデリング要素のいくつかに使用されるアイコンを定義することでした。表記のデフォルト記号が適切でない場合、さまざまな言語のコンセプトをツールバーおよびブラウザの小さいアイコン記号で区別しやすくしましたが、これらは 1 時間以内に定義されました。

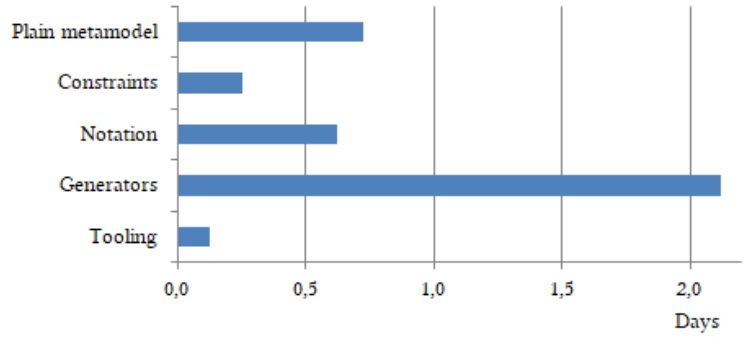

図 2　IoT ソリューションのパート別の作成に要する人日

3.2　ArchiMate

二つ目の言語は The Open Group の ArchiMate です[28, 29]。ArchiMate の主な用途は、エンタープライズ系のアーキテクチャをモデルで表現して、コミュニケーションを取り、分析することです。この言語は、ビジネスと IT の両方を含むさまざまな利害関係者からの懸念を認識することを目的としています。設計者がモデルを作成するにしても、誰もがモデルを理解してコメントすることができるものです。この目的のために ArchiMate は推奨されるアイコンや他の表記要素と共にエンティティとそれらの関係のセットを提供します。図 3 は保険金請求に関連するビジネスプロセス、サービス、および役割を指定する際の、実装されたモデリングソリューションにおける ArchiMate の使用方法を示しています。

ArchiMate モデルは、フロー、トリガー、およびアクセス関係を使用したプロセスモデリングとともに、関連付けと集約を使用した従来のデータモデリングの組み合わせです。ArchiMate の特別な機能は、モデルの階層化を提供し（ビジネス、アプリケーション、テクノロジー層）、層を越えて特別な関係の要素を関連付けることです。たとえば、図 3 では、黄色の上部がビジネス層を示し、下部の 4 つの青い要素がアプリケーション層を示しています。ここでは、顧客データ管理と支払い処理の 2 つのアプリケーションサービスが、ビジネス層の要素とサービング関係を持っています。この言語はコンセプトの背景に厳密な意味論を持たず、モデルを作成するための実際の手引きを提供しません。すなわち ArchiMate では、同じ考えをさまざまな方法でモデル化することができます。

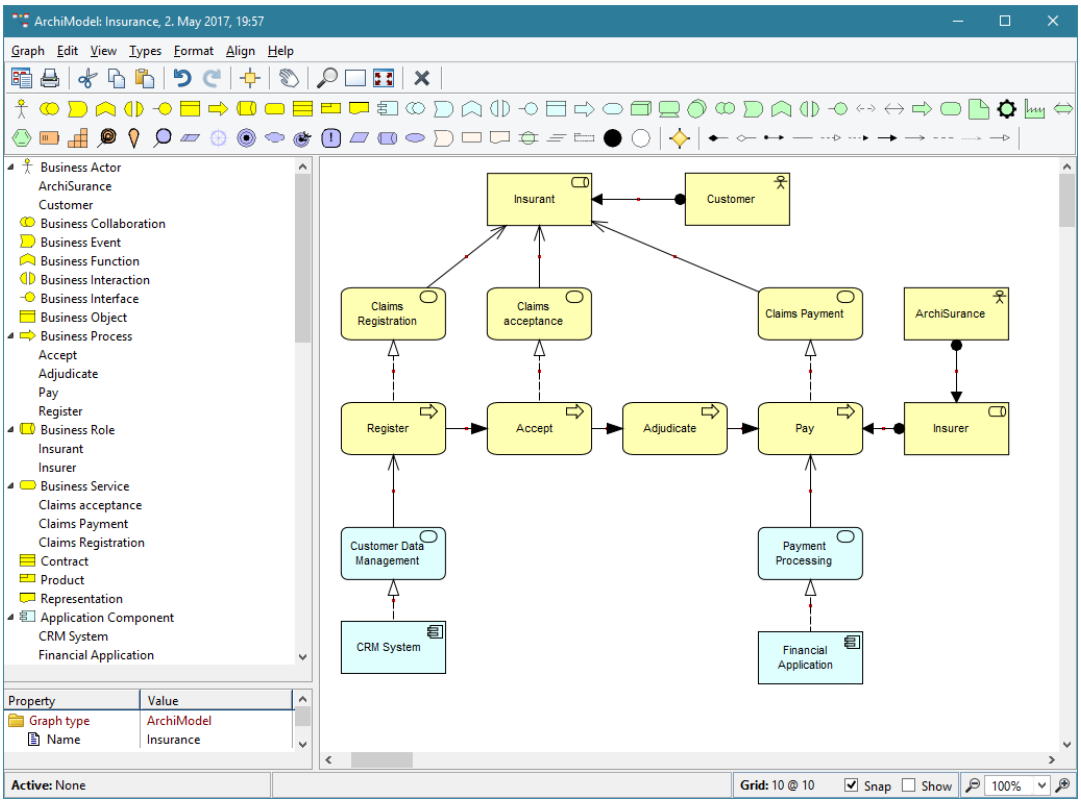

図 3　ArchiMate の使用

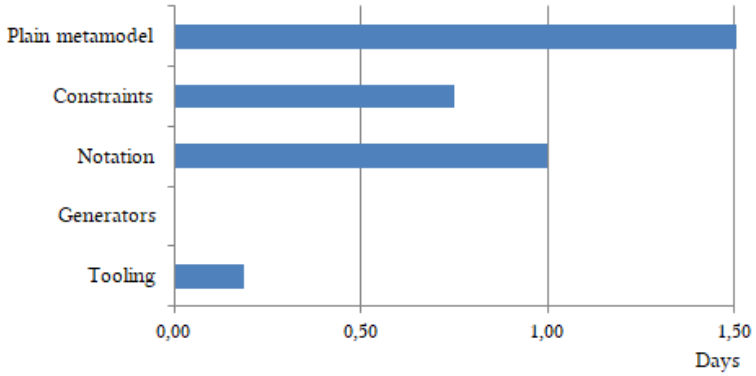

図 4　ArchiMate モデリングサポートのパート別の作成に要する人日

言語は階層化に加えて、動機、リソース、および作業結果に関連する要因を説明するためのさまざまな追加のコンセプトを提供します。また図に記載されるモデルは急速に（あっという間に）大きくなる可能性があり、また、さまざまな利害関係者に異なったビューを表示する必要があるため、ArchiMate はモデルに対して一連の特定の視点を提案します。

ArchiMate モデリングサポートの実装には明確な仕様が得られたため、IoT の場合よりも異なる選択や表記記号を考慮する必要性が少なくなりました。また、ArchiMate はコード生成をサポートしていないため、モデルチェック以外にジェネレータを実装する必要はありませんでした。当初、ArchiMate の実装は、プロセスモデリング用の BPMN やデータモデリング用の ER など、既存の MetaEdit+のメタモデルと統合して作成されました。ケーススタディの目的のために、これらの拡張と統合は除外され、データ収集は純粋に ArchiMate のパーツにのみ焦点を合わせました。

3.2.1 実装された DSM ソリューション部品　実装されたメタモデルは 125 の要素を含みます。これらは ArchiMate の仕様に従いモデリング要素を分類するために使用される 24 の抽象要素を含みます。メタモデルの抽象要素は、言語ユーザーには直接表示されず、モデリングサポートの作成に必須ではありません。 ArchiMate の仕様では、制約を定義する際にそれらを使用しません。

制約は ArchiMate の特筆すべき機能であり、特に膨大な数の制約です（[29]、付録 B2）：仕様には 10,000 を超えるルールがあり、ほとんどすべてが 11 のリレーションシップタイプのうちどれが 61 のオブジェクトタイプに適正であるかを定義します。

アーキテクチャモデルはコードの生成には使用されないため、テキスト記述をモデルに入力する方法や、データが必須の場合でも、実際の制約はありません。IoT の場合と同様に、モデリング中に統合されたヘルプを提供できるように、言語コンセプトの説明がメタモデルに追加されました。これらは、そのオブジェクトやリレーションシップなど、すべての主要なモデリング要素に関するガイダンスを網羅しています。

表記部分については、89 個のシンボルが定義され、元の仕様とよく似せています。これらのシンボルのほとんどは、オブジェクトの種類を示す特別なアイコンと階層の推奨カラーを含む長方形です。 ArchiMate の仕様には推奨アイコンが含まれているので、これらもツールバーやブラウザに表示するようにしました。全部で 77 のアイコンが定義されています。

ジェネレータはチェック用のもののみが定義され、メタモデルでは定義できなかったいくつかの制約に対処しています。これらには、ジャンクションコンセプトを使用して 3 つ以上の要素間のリレーションシップを表現する方法、モデリングオブジェクトをグループ化する方法、複合関係に関連する要素が正しいことを確認する方法が含まれています。これら 3 つのジェネレータは小さなものです（MetaEdit+ MERL 言語で 42 行）。

これらの定義に基づいて、MetaEdit+はアーキテクチャモデリングのコラボレーションのためのエディタ、ブラウザ、モデル管理ツール、そしてその他の期待されるモデリング機能を提供します。ビューの管理に関しても、同じモデルまたは単一のダイアグラムに対して異なるビューを持たせる MetaEdit+の機能を利用できます。同じ要素を異なるビュー間で共有することもできます。このようにして、異なる利害関係者に対してデータを適切に表示することや、特定の層をモデルから隠すことができます（たとえば、焦点がアプリケーション層にあり、他の層を隠したい場合）。この ArchiMate 用の DSM 言語は https://github.com/mccjpt/ArchiMate から入手できます。

アーキテクチャモデルを Word または HTML としてレポートおよび公開するために、MetaEdit+の既存の定義済みジェネレータを使用することができます。また MetaEdit+に標準装備されるモデルチェックやメトリクス計算用のジェネレータも ArchiMate モデルで利用できます。

3.2.2　DSM の実装努力　ArchiMate モデリングソリューションの実装には、約 4 人日かかりました。この取り組みは、図 4 に示すように、言語定義のさまざまな部分に分割されました。メタモデルは、4 つのセッション（合計 12.1 時間）で定義されました。リレーションシップに対する制約の数は膨大で、合計で 10,760 です。それらをメタモデルに手動で追加するのは大変な作業であるので、MetaEdit +のジェネレータを使って言語定義のこの部分を自動化しました。リレーションシップのテーブル（[29]、付録 B）の定義が解析され、バインディングルールがメタモデルに生成

されました。モデル構造の階層化を扱う残りのいくつかのルールは手動で定義されました。単純なテキストのルール情報を解析して、MetaEdit+メタモデルの XML インポートフォーマットで制約を出力するジェネレータを定義する作業は 6 時間でした。

この調査では言語開発の最初の作成段階に焦点を当てていますが、言語制約の自動生成については言語の保守段階にも関わってくる可能性があります。たとえば、ArchiMate 言語は、Open Group による部分的な定義で作成されていますが、その正しさを保証するためにテストや参照データを使用していません。そのため、以前のバージョンには多くのエラーがありました。たとえば、最新のメジャーリリースである ArchiMate 3.0 [28]では、リレーションシップの定義に何百ものエラーがあり、バージョン 3.0.1 [29]ではこれらの修正が行われました。このような場合でも、制約の自動生成機能により、ジェネレータを実行するだけで新しいバージョンのすべての制約をメタモデルに更新することができます。

オブジェクト、リレーションシップ、ロール、アイコンのシンボルをカバーする表記は、8 時間で定義されました。非常によく似た表記要素を多数作成しないようにするために、MetaEdit+が提供するシンボルライブラリとテンプレートメカニズムを使用しました。図 5 は「Technology Interface」要素の表記の定義例です。MetaEdit+の Symbol Editor で選択された右上隅のテンプレート（点線部分）は、シンボルライブラリから正しいアイコンを取得し、それを残りのシンボルで拡大縮小しないように保ちながら正しく配置します。同じアイコンは ArchiMate の他のインターフェース要素、すなわち「Application Interface」、「Technology Interface」および抽象「External Active Structure Element」にも使用されます。表記を定義するためのこのパターンは、そのさまざまな機能、サービス、コラボレーションなどの他の ArchiMate 要素に対しても繰り返されました。全部で 29 の表記記号が再利用されました。これにより、表記の作成が容易になるだけでなく、アイコンが一貫した方法で使用されるようになりました。ジェネレータがなく、モデリング要素用に作成されたアイコンをツールバーのアイコンとして直接使用できるため、追加の開発作業はほとんどなく、合計 1.5 時間でした。

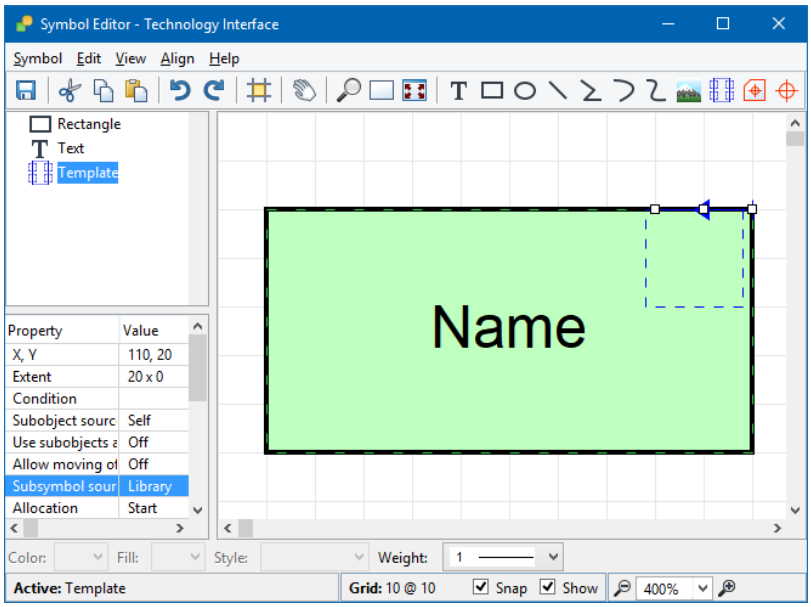

図 5　Technology Interface に対する表記の定義

4　業界の事例報告における開発の取り組み

我々の 2 つのケーススタディは現実的な規模の言語に焦点を当てましたが、実製品開発では無いので、組織内のコミュニケーションや、リソース割り当て、その他の要因などの影響がない環境で開発されました。これを他の業界事例の開発努力と比較するために、他の言語エンジニアによって公開されたデータ － 論文またはプレゼンテーション － を調べました。特定の課題に焦点を当てた言語であり、かつその努力が明確に述べられている論文または関連する発表を選びました。そのため、企業が複数のジェネレータを開発した場合は除外されています。ただし、比較を可能にするためにデータを分離していない限りです。

図 6 は、以下に紹介される各事例の実施努力を要約しています。すべての場合において、言語開発とその利用は同じツール MetaEdit+を使って行われました。これによりツールによる影響を除外することができます。8 つのケースのうち 4 つでは、言語開発は各社の言語エンジニアによって行われ、DSM の作成時間も測定されました。 3 つのケース（暖房リモコン、軍用無線システムテスト、血液分離器）では、言語エンジニアリングの主な作業は外部コンサルタントによるものでした。1 つのケース（音声制御）では、言語エンジニアリングは MetaCase 社のコンサルタントによって行われました（ただし著者ではありません）。私たちのケーススタディと同様に、すべての業界事例で言語エンジニアリングチームの合計は 1 人か 2 人でした。

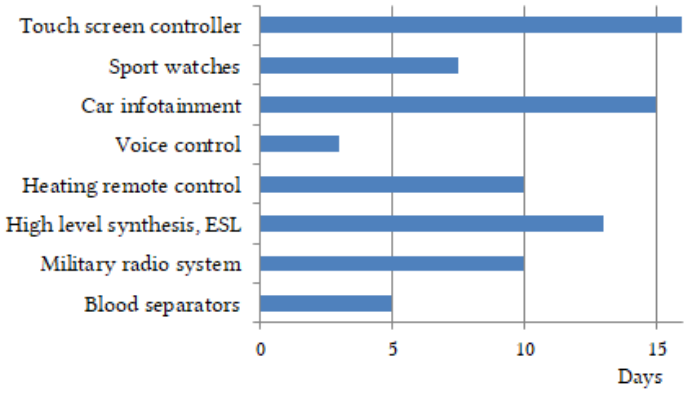

図 6　事例ごとの DSM ソリューション作成に要した人日

4.1　業界事例のレビュー

4.1.1　Panasonic のタッチスクリーンコントローラ[25]　この言語はホームオートメーション用の組み込み機器のアプリケーション開発用です。ジェネレータは作成されたモデルを読み、Linux ベースのタッチスクリーン用の C と HTML を生成します。パナソニックは、DSM ソリューションのさまざまな要素への取り組みを区別しませんでしたが、開発の総工数を 16 人日と報告しました。また一方、後日同じモデルを使用したマイクロコントローラ用の 2 つ目のジェネレータの開発には 3 人日であったことが報告されています。このジェネレータは C コード生成に伴って、ビルド/フラッシュ書込み/実行といった自動化処理も生成します。

論文：https://www.fuji-setsu.co.jp/files/dsm07safa.pdf

4.1.2　Polar 社のスポーツウォッチ[14]　この DSM は、スポーツウォッチの組み込みアプリケーション開発用です。モデルは、アプリケーションの UI、ナビゲーション、およびローカライゼーションを指定して、最終製品のコードの約半分の C コードを生成します。Polar 社からの報告では、DSM ソリューションの開発に 60 人時間を費やしました。

事例スライド p12~18：https://www.fuji-setsu.co.jp/files/DSM_IndustryCases2012.pdf

4.1.3　OEM によるカーインフォテインメントシステム[2]（ライプツィヒでの講演）　自動車メーカーは、インフォテインメントシステムのエンドユーザーアプリケーションを仕様化するための DSM ソリューションを開発しました。モデルは静的 UI 要素、UI ロジック、ナビゲーション、サービス（ラジオなど）の呼び出しなどをカバーします。ジェネレータはシミュレーションとコンセプト設計のための Java を生成します。OEM は DSM ソリューションを作成するために合計 3 人週間を投資しました。言語に 1 週間、ジェネレータに 2 週間でした。

関連情報：https://www.metacase.com/cases/autoinfo.html

4.1.4　ホームオートメーションのための音声制御システム[11]　開発者はエネルギーおよびホームオートメーション分野の機器間通信に的を絞りました。この DSM 言語は、電話回線を介した音声コマンドと、それによる制御ロジックの指定に焦点を当てています。DSM 言語は、MetaCase 社のコンサルタントとドメイン分析のためのドメインの専門家による 2 日間のワークショップで、3 人日の工数で実装されました。ワークショップの初日には、コード生成をするには汎用的すぎる言語が作成されたため、時間のかかるプレッシャーの下で最初からやり直す必要がありました。 2 回目の試行はモデリング言語に 1.7 時間かかり、8 ビットマイクロコントローラ用のアセンブラコードを生成するジェネレータの実装はわずか 2 時間程度でした。

関連情報：https://www.metacase.com/cases/microcontroller.html

4.1.5　Ouman 社の暖房システムの遠隔操作[23]　これは携帯電話からコントロールするアプリケーションを開発する DSM 言語で、ジェネレータは Python を生成します。 外部コンサルタントが、Ouman の 1 人と共同でこのソリューションを開発しました。コンサルタントが単独で実装を行い、Ouman の担当者はそれをテストしてフィードバックを提供しました。コンサルタントによる開発努力は 2 人週でした。

事例スライド p19~24：https://www.fuji-setsu.co.jp/files/DSM_IndustryCases2012.pdf

4.1.6　Profound 社での仮想プラットフォーム用の高位合成[22]　DSM ソリューションは、画像処理システムのブロック構造を指定し、高位合成用の SystemC を生成します。 最初のバージョンを作成するために、Profound 社はメタモデルに 4 日、表記に 3 日、SystemC ジェネレータに 5 日を要しました。 実装は、高位合成のドメインに精通する人によるものです。

講演資料：https://www.fuji-setsu.co.jp/files/SystemC_Japan2012_PDT.pdf

4.1.7　Elektrobit 社の軍用無線システムのテスト[24]　VoIP ネットワークと関連端末をテストするために、この DSM ソリューションはテストケースと大規模なテストロジックを指定します。生成されたコードは TTCN3 テストスクリプト言語です。DSM の開発努力は 2 人週でした。また追加の 1 週間で、テスト環境からモデリングツールへのフィードバック機能を実装しました。これによりモデルに注釈を付けてテストの実行を視覚化しました。言語のテストは、テストケースとして既存の参照ソリューションを作成する別の人によって、後で行われました。

事例スライド p25~33：https://www.fuji-setsu.co.jp/files/DSM_IndustryCases2012.pdf

4.1.8　血液分離器[5]　これは医療機器製造業者の血液分離器のプロダクトライン用の DSM 言語です。 ハードウェアとソフトウェアを含む開発ソリューションの一環として、MetaCase 社のパートナーのコンサルタントがこのプロダクトラインのモデリング言語を開発しました。IEC 61131 プログラムコードとコンフィグレーション用のコードを生成するジェネレータが開発されました。この DSM ソリューションの作成は 1 人週の工数であったことが報告されています。

関連情報：https://www.metacase.com/cases/BloodSeparationMachine.html

4.2　ケーススタディと業界事例の比較と結合

2 つのケーススタディと業界事例の主な違いは、業界事例ではドメインの知識が既にあったということです － 少なくとも言語エンジニアは、当該ドメインの問題空間と解決空間（実装）の経験がありました。これに対して、ケーススタディのほうは各ドメインに不慣れでしたが、言語エンジニアリング全般、特に MetaEdit+に精通している経験豊富な言語エンジニアによって行われました。また、ケーススタディは既存の言語コンセプトのセットから始めることができましたが、業界事例ではまずドメインのコンセプトを特定する必要がありました。

業界事例では、DSM ソリューションの作成に平均 2 週間弱(9.8 日)の開発工数が報告されています － 3 日～16 日の範囲で、標準偏差は 4.6 日。我々の 2 つのケーススタディは 4 人日の開発工数であり、業界事例に収まる範囲です。社内リソースのみを使用する業界事例は平均 12.6 日かかりましたが、外部コンサルタントを使用するケースは平均 7 日でした。MetaCase 社の言語エンジニアが関わった 3 ケース（業界事例 1 つと 2 つのケーススタディ）は平均 3.4 日でした。

これらの数字から言語開発者の経験差による違いを考慮する必要があると考えられます。そしてドメインに依存しない要素があって、一度習得すれば次回から効率的にできそうです。ただ、どこまでが言語開発の一般的な学習で、どの程度 MetaEdit +の学習に必要かについては疑問が残ります。いずれにせよ、言語の開発と MetaEdit+の両方に初心者であっても、平均 2～3 人週でドメイン固有のモデリング言語を作成できます。

今回得られた業界事例の殆どは、開発工数を DSM ソリューションのさまざまな部分に分割していませんが、1 つのジェネレータを有する DSM ソリューションを実装する場合、そのジェネレータの構築には全体の 40～60%を要しています（高位合成で 42%、音声制御システムで成功した 2

回目の試行で 54%、IoT では 55%）。ジェネレータを除くと、表記は全体の約 30〜45%を占めました（ArchiMate では 29%、IoT では 36%、高位合成では 43%）。

さまざまなフェーズと成果物を見ると、おそらく事例間の最大の違いは、ドメイン分析、言語の境界とレベルの決定、および言語の主なアイデアの思いつき、など DSM 開発工程の最も早い段階にあります。これらの作業のために別々に工数を分析されることはありませんでした。そして確かにそれらはしばしばメタモデリングと表記のフェーズにまたがって広がっています。音声制御システムの場合に見られるように、比較的経験豊富な言語エンジニアでさえ誤った選択肢を選び、最初からやり直すことを強いられる可能性があります。これらの初期段階では、メタモデルを迅速かつ段階的に構築することを可能にするツールの能力が求められます。物事を具体化することで時間の浪費が減り、問題の早期発見とさまざまなアプローチによる実験が可能になるからです。

言語の規模は、必要な作業量に対して殆ど影響しません。残念ながら、ほとんど公にされていませんが、3 つの業界事例からの言語サイズに関するデータを得ています。もっと多くのデータが必要になるでしょうが、サイズに工数が比例することは無く、より少なくなるようです。最大規模が UML の約 2 倍の言語であっても、スケーラビリティや複雑さの問題にぶつかって遅くなるということは確かにありません。大規模言語では、言語をさらに拡張するためのパターンが開発されているようです。新しいコンセプトを追加するのは簡単なことが多く、類似した既存のコンセプトのセットのもう 1 つのメンバーになります。

5　取り組みを比較している他の研究

私たちは様々な DSM ソリューションの開発努力を比較する体系的な研究を多くは知りません。大抵の場合、開発努力を報告している研究は単一のケースに焦点を合わせており、ドメイン、言語、コード生成などのニーズおよびツール、等々の違いは個々のケースにわたって比較することを難しくしています。

比較のある場合でも、直接工数を集めることはありませんが、異なる言語のコード行数などの代用物を介します。例えば、[18]では、言語エンジニアが、言語定義メカニズムの違いを適用して、同じモデリング言語を 2 回実装しました。報告された言語仕様から、OCL 制約を使用して UML プロファイルを定義するには、MetaEdit+が提供する制約を使用したメタモデルでモデリング言語を実装するよりも 2 倍多くの制約仕様が必要です。

同様に、いくつかのツールに同じ一連のタスクを与え、それらの仕様の長さをコード行で比較しました[7]。残念ながら、コード行数はツールごとに異なった言語に対するものであり、ツールによってタスクのどの部分が含められるか異なるため、比較の正確性と読みやすさが低下します。しかしながら、[12]のさらなる分析がコード行数に対して完了したタスクをグラフ化することによっていくつかの情報を回復することで、データを公開することの重要性が強調されました。

一般的に見慣れないさまざまな言語のコード行数を比較するよりも少し優れているのは、既知の言語のコード行数を時間の経過の代用として使用する場合です。これは[10]で取られたアプローチであり、Java で知られている Eclipse GEF の事例を取り、COCOMO を使ってその 10,000 行のコードに基づいてその開発努力がおよそ 13 人月（2,000 時間）であると見積もりました。比較として、同じ言語は MetaEdit+では 1 時間で実装されました。内訳は、抽象メタモデルに 15 分、表記

に 45 分でした。これは、開発時間をその部分に分割した数少ないケースの 1 つです。

時間はコード行数よりも工数に対する良い尺度です。 De Smedt [26]は、同じ単純な言語とその変換・シミュレーションのサポートを彼の部門の AToM3 では 13 時間で、MetaEdit+では 11 時間で構築しました。この実験では、タスクに適したツールを選択することの価値が示されました。AToM3 はシミュレーションに焦点を当てているため、API を使用しなければならなかった MetaEdit+よりもその部分が簡単になります。同様に、Poseidon を使用しようとする試みは、変換（コード生成などの用途に必要な）のサポートがないために早々にやめなければなりませんでした。

[21]では、言語設計中に下された決定を評価することに的を絞って、2 つの異なる DSM ソリューションが倉庫自動化ドメインのために作成されています。モデリング言語の作成に加えて、IEC 61131-3 のソースコードと自動化システムの視覚化を含む、完全な自動化ソフトウェアを作成するための 2 つのジェネレータが定義されました。これらの DSM ソリューションは両方とも MetaEdit+で定義されましたが、異なる言語設計パターンと意思決定に従っていました。結果として得られた DSM ソリューションは、同じ倉庫システム機能を開発するために適用されました。実装は、DSM や自動化システムの分野での経験がない学生によって行われました。学生の仕事として完全な DSM ソリューションを開発するための工数は平均 33 人日でした。

最後に、ツールが DSM ソリューションを作成するのに必要な努力に著しい違いをもたらすことを調査は示します。実証的研究[13]は、同じ言語の一部である BPMN が 5 つの異なるツールで実施された対照実験室研究を適用しました。したがって、この研究は既存の言語仕様に基づいて ArchiMate を実装する場合に似ています。この調査では、使用したツールによって開発作業に 0.5 日から 25 日の間の大きな違いがあることが示されました。開発工数が最短であったのは MetaEdit+のバージョン 4.5 [16]を使用した場合で、最も長くかかったのは Eclipse Graphical Modeling Framework 2.4.0 [8]によるものでした。

5.1 他の研究とケーススタディ結果の統合

[13]でおそらく最も注目に値するのは、開発工数が 2 番目に短かったツールでも、一番短かった MetaEdit+を使った場合よりもすでに 1 桁遅いということです。考えられる要因の 1 つは、MetaEdit+を使用した実験対象が、他のツールを使用した場合よりもはるかに優れた言語エンジニアであったということの可能性です。この資料で紹介した我々のケーススタディでは、最も経験豊富な言語エンジニアと初めての人との間で 3.75 の係数を示しています。これをテストするために、本論文の著者の 1 人が同じ MetaEdit+バージョンで同じ BPMN サブセットを実装しました。この再実装には 45 分かかり、最初と比べて 5 倍以上速くなりました。これによって、元の実験対象を他の初めての人と同等のスキルレベルにできました。

同様のスキルレベルの言語開発者によって同じ言語が異なるツールで作られたとき[13]、工数は 50 倍異なりました。同じツールが異なる言語を作るために使われたとき、工数は 5 倍に変化しました 。ここで取り上げる事例の範囲 - ドメインと言語のサイズと複雑さ、ジェネレータの必要性、および言語実装者の経験（ツール、言語作成全般、ドメインの問題空間と解決空間の両方）を考えると、5 の係数は比較的控えめに見えます。

ツールを基準にして 50 倍、他のすべての要素を合わせて 5 倍、ツールの選択によって開発作業

に最大の違いが生じると言えるでしょう。この工数は言語エンジニアの時間のコストに変換されます。これは産業界ではツール価格の要素をはるかに上回ります。これは、ドメイン固有の言語を作成するチームはツールの選択に特別な注意を払うべきであり、リソースと時間の予算を組むときにはそのことに注意する必要があることを示しています。

我々は開発努力、使用された時間、チームの大きさ等に関するデータを報告する、業界での他の事例研究と言語開発への取り組みに期待します。取り組みを収集および分析するための適切な測定基準を確立することは、特に産業界の場合には困難です。理想的には、研究は言語作成のさまざまな要素や段階（抽象構文、制約、表記、ジェネレータ、およびツーリング）に関するデータを収集されるでしょう。

6 結論

詳細なケーススタディと業界事例という 2 つの補足資料を組み合わせて、10 のケースにわたって完全な DSM ソリューションを作成するための取り組みを比較しました。ケーススタディでは、DSM ソリューションのさまざまな部分を作成するためにどれだけの労力が費やされたかについての詳細なデータを提供しました。 2 つのケーススタディ言語は異なりますが、収集されたデータには類似点があります。メタモデルで抽象構文を作成することは、表記に使用される具象構文を作成することよりも時間がかかるようです。

このツール（や他のもの）を使用した業界事例の大部分は、その努力をまったく示していませんでした。しかし、MetaEdit+を使用して DSM ソリューションを作成するための努力はそれほど多くないことを示しています。これは、3 から 15 人日、平均 10 人日の範囲です。これは、4 人日の開発による 2 つのケーススタディと一致しています。DSM ソリューションはさまざまなドメインに対応しており、またさまざまな経歴や経験レベルを持つ言語エンジニアによって作成されていますが、取り組みの規模は DSM ソリューション間で非常に似ています。私たちのケーススタディと同様に、実際の実装はすべての業界事例でも 1 人か 2 人によって行われました。これは大規模なチームが言語開発に必要ではないことを示しています（少なくとも MetaEdit+を使用する限り）。また当然のことですが、言語および DSM ソリューションのテストおよび段階的(インクリメンタル)な開発ではより多くの言語ユーザーが歓迎されます。

残念ながら、ほとんどの業界事例では、開発作業を DSM ソリューションのさまざまな部分に分割した分析はできませんでした。業界の事例から入手可能なデータは、ジェネレータ機能を持つ DSM ソリューションを実装する場合、ジェネレータの構築にはモデリング言語の作成（メタモデル、制約、および表記）の約 2 倍の労力がかかることを示しています。 取り組む分野と言語エンジニアの経験に違いがある場合など、10 の事例のみで開発工数を一般化するのは十分ではありませんが、それでも投資を大きくする必要はありません。

DSM ソリューションが利用可能になると、言語ユーザーはモデルを作成してコードをすばやく生成することができるようになりました。DSM の高い生産性は優れた投資収益率を提供します。そして、ROI 計算のためのデータが示すように、言語を開発するのに費やされた努力はすぐに返済されます[14, 21, 24, 25]。

ここでは言語作成段階に焦点を合わせましたが、すべてのソフトウェア開発と同様に、メンテナ

ンスは通常、総作業量の点で大きいと認識しています。ただし、言語、ジェネレータ、およびツールを更新するための作業は、通常、最初のフェーズで言語を作成するための作業に比例します。ただし、大きな違いは、メンテナンス中に言語を変更すると、既存のモデルを新しい言語バージョンに移行する必要があることです。そのために必要な作業はツールによって大きく異なります。ライフサイクル全体にわたるその違いの影響を定量化するには、将来の実証的な調査が必要です。

経験的な研究手法を言語工学や言語開発に応用した将来の研究、さらには言語、ジェネレータ、ツールを生み出す業界事例からの分析データを歓迎します。研究者は DSM ソリューションの一部の同じ分類法に従ってこの研究を繰り返し検証し、その努力を報告することができます。研究の範囲は、他の分野や他の種類のモデリング言語に取り組むことによっても拡大させることができます。我々が 1 つのツールに焦点を当てた結果、さまざまなツールが実装作業に大きな影響を及ぼしていることが調査によって示されました。さまざまなツールを取り上げたり、特定のツールで DSM ソリューションのどの部分が特に速くなったり遅くなったりするかを特定するためのさらなる研究を歓迎します。

参考文献

[1] Ronan Barrett. 2015. 5 Years of 'Papyrusing', Ericsson Modeling Days, Kista, Sweden, November 9-10, 2015. Retrieved from https://docs.google.com/presentation/d/1nR6GRBsS2Ad3OQf8ldMO8MydFEeWFsFd8WqjQ23Bz8s

[2] Carsten Bock. 2006. Visuelle domÃď nenspezifische Sprachen — Der Schlüssel zur modellgetriebenen Entwicklung von Mensch-Maschine-Schnittstellen?, Model-Driven Development and Product Lines: Synergies and Experience, Leipzig, Germany, October 19-20, 2006.

[3] Francis Bordeleau. 2014. Papyrus and Open Source Modeling — Status, Strategy, and Plan. Presentation at Ericsson Modeling Days, Kista, Sweden, 4 November, 2014

[4] N. Brouwers, R. Hendriksen, K. Kahraman, J. Kouwer, and J.-P. Tolvanen. 2016. Industrial use of domain-specific modeling, DSM workshop, SPLASH. Retrieved from http://dsmforum.org/events/DSM16/

[5] Verislav Djukić, Aleksandar Popović, and Juha-Pekka Tolvanen. 2014. Using domain-specific modeling languages for medical device development, Embedded.com, March 08, 2014.

[6] DSMForum.org. 2018. http://dsmforum.org/cases.html

[7] S. Erdweg, T. van der Storm, M. VÃűlter, M. Boersma, R. Bosman, W.R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P.J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. 2013. The State of the Art in Language Workbenches: Conclusions from the Language Workbench Challenge. In Proceedings of the Software Language Engineering conference, Springer, 2013.

[8] Richard Gronbach. 2009. Eclipse Modeling Project. A Domain-Specific Language (DSL) Toolkit. Addison Wesley.

[9] Steven Kelly, Kalle Lyytinen, and Matti Rossi. 1996. MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In Proceedings of CAiSE'96. Springer.

[10] Steven Kelly. 2004. Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM. In Proceedings of the OOPSLA and GPCE Workshop on Best Practices for Model Driven Software Development. http://www.softmetaware.com/oopsla2004/mdsdworkshop.html

[11] Steven Kelly and Juha-Pekka Tolvanen. 2008. Domain-Specific Modeling: Enabling Full Code Generation. Wiley.

[12] Steven Kelly. 2013. Empirical Comparison of LanguageWorkbenches. In Proceedings of the 2013 ACM Workshop on Domain-Specific Modeling. http://dsmforum.org/events/DSM13/

[13] A. El Kouhen, C. Dumoulin, S. Gérard and P. Boulet. 2012. Evaluation of Modelling Tools Adaptation. CNRS

HAL hal-00706701. http://tinyurl.com/gerard12

[14] Juha Kärnä, Juha-Pekka Tolvanen, and Steven Kelly. 2009. Evaluating the use of domain-specific modeling in practice. In Proceedings of the 9th OOPSLAworkshop on Domain-Specific Modeling. http://www.dsmforum.org/events/DSM09/Papers/Karna.pdf

[15] M. Mernik, J. Heering, and A. Sloane. 2005. When and How to Develop Domain-Specific Languages, ACM Computing Surveys, 37, 4.

[16] MetaCase. 2006. MetaEdit+ 4.5 User's Guide. http://www.metacase.com/support/45/manuals/

[17] MetaCase. 2017. MetaEdit+ 5.5 User's Guide. http://www.metacase.com/support/55/manuals/

[18] K. Mewes. 2009. Domain-specific Modelling of Railway Control Systems with Integrated Verication and Validation. Ph.D. thesis. University of Bremen.

[19] P. Mohagheghi, W. Gilani, A. Stefanescu, M. Fernandez, B. Nordmoen, M. Fritzsche. 2011. Where does model-driven engineering help? Experiences from three industrial cases. Software and Systems Modeling 12, 3, 619-639.

[20] Object Management Group. 2017. Unified Modeling Language Version 2.5.1.

[21] C. Preschern, N. Kajtazovic, and C. Kreiner. 2014. Evaluation of Domain Modeling Decisions for Two Identical Domain Specific Languages, Lecture Notes on Software Engineering 2, 1 (Feb. 2014). DOI: https://doi.org/10.7763/LNSE.2014.V2.91

[22] Profound. 2018. Construction of cooperative design environment of software / hardware using single-threaded ESL tool. Retrieved April, 2018 from http://www.profound-dt.co.jp

[23] Olli-Pekka Puolitaival. 2011. Home Automation DSL Case, Presentation at Code Generation Conference, 2011.

[24] Olli-Pekka Puolitaival, Teemu Kanstrén, Veli-Matti Rytky, and Asmo Saarela. 2011. Utilizing domain-specific modelling for software testing, 3rd Int. Conference on Advances in System Testing and Validation Lifecycle.

[25] Laurent Safa. 2007. The Making of User-Interface Designer, a Proprietary DSM Tool. 7th OOPSLA Workshop on Domain-Specific Modeling. http://www.dsmforum.org/events/DSM07/

[26] P. De Smedt. 2011. Comparing Three Graphical DSL Editors: AToM3, MetaEdit+ and Poseidon for DSLs. University of Antwerp. http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/201011/projects/Philip.DeSmedt/report/report_PhilipDeSmedt.pdf

[27] J. Sprinkle, M. Mernik, J.-P. Tolvanen, and D. Spinellis. 2009. What kinds of nails need a domain-specific hammer?. IEEE Software, (July-Aug, 2009).

[28] The Open Group. 2016. ArchiMate 3.0 Specification.

[29] The Open Group. 2017. ArchiMate 3.0.1 Specification.

[30] The Open Group. 2017. Architecture Tool Certification: ArchiMate 3 Conformance Requirements.

[31] Juha-Pekka Tolvanen. 2016. Applying Test-Driven Development for Creating and Refining Domain-Specific Modeling Languages and Generators, In Proceedings of 16th Workshop on Domain-Specific Modeling, Amsterdam.

[32] J. Whittle, J. Hutchinson, and M. Rouncefield. 2014. The State of Practice in Model-Driven Engineering, IEEE Software 31, 3 (May-June, 2014)

[33] Markus Voelter. 2013. DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. CreateSpace