PhD Thesis

The Design and Implementation of the Management Action Language and the Life Cycle of Other DSLs

Holger Stadel Borum

Advisor: Peter Sestoft
Dept: Computer Science
Submitted: July 29, 2022

IT UNIVERSITY OF COPENHAGEN

Is [TypeScript's] type system sound?

No.

Have you tried to prove that [TypeScript's] type system is sound?

No.

Do you want to?

No.

Q&A session
Anders Hejlsberg

Abstract

A domain-specific language (DSL) is a programming language designed for use in a particular problem domain. Typically, a DSL is more convenient, expressive, safe and usable within its intended domain, in return for being less general and less useful in other domains. Hence, DSLs are applied across many domains to alleviate software engineering problems such as programmer productivity, code efficiency, software variability, and software quality. Due to their diverse application domains and purposes, DSLs take many different forms, and DSL creators encounter diverse challenges.

In this thesis, we explore the design, implementation, and lifecycle of DSLs. Specifically, we develop a rather non-trivial DSL called Management Action Language (MAL) for the pension and life insurance domain together with a major software vendor. Through a highly introspective process, we build on the lessons from this experience to draw conclusions about DSL design, implementation, and evolution. We complement this work by sending a questionnaire to the creators of other DSLs to survey the creators' experience.

First, we explore human-centred design and co-design of DSLs to ensure that our DSL, or any DSL, is usable by its intended users. We use our experiences with designing MAL to propose a two-phase human-centred design method and show that non-programming experts can be used generatively in a DSL design process.

Second, we explore DSL implementation, focusing on run-time performance and safety. Within the context of MAL, we show how we implemented a code generator that creates more efficient code than comparable handwritten code while providing compile-time guarantees on initialisation and reserve preservation.

Third, we survey the established practices in the lifecycle of DSLs through a questionnaire sent to designers of historical DSLs. We focus on different phases in a DSL's lifecycle and show, among other things, that (a) we find no correlation between the level of user involvement in the design process and the level of programming experience of users of DSLs and (b) that most DSLs evolve after creation, and that handling this evolution affects the practical success of a DSL.

Finally and separately, within the context of so-called Dynamic Condition Response (DCR) graphs, a kind of DSL for describing workflows in organisations developed by Hildebrandt and Mukkamala, we describe our work towards providing static secrecy guarantees when the workflows adversarial parties that may seek to infer actor decisions intended to be private.

Resumé

Et domæne-specifikt sprog (DSL) er et programmeringssprog designet til brug i et afgrænset domæne. Typisk bytter et DSL tilgængelig, udtrykskraft, sikkerhed og brugervenlighed i dets domæne for at anvendelig i andre domæner. Derfor bliver DSL'er brugt på tværs af mange forskellige domæner til at løse problemstillinger i softwareudvikling som f.eks. programmørers produktivitet, effektiv kode, software variabilitet og softwarekvalitet. DSL'ers mangeartede anvendelser og formål betyder, at skabere af DSL'er støder på lige så mangeartede udfordringer.

I denne afhandling udforsker DSL'ers design, implementering, og livscyklus. Vi udvikler et ikke trivielt DSL kaldet Management Action Language (MAL) for pensions og livsforsikingsdomænet sammen med et større softwarefirma. Vi bruger vores erfaringer fra en introspektiv proces til at drage konklusioner om DSL-design, implementering og evolution. Vi komplementerer dette arbejde med en spørgeskemaundersøgelse sendt til skabere af andre DSL'er, som undersøger andres DSL-erfaringer.

Først undersøger vi brugen af *human-centred design* og *co-design* af DSL'er til at sikre, at et DSL kan bruges af dets målgruppe. Vi bruger vores erfaringer med at designe MAL til at foreslå en tofaset human-centred designmetode, og vi viser at eksperter uden programmeringsbaggrund kan bruges skabende i en designproces.

Derefter undersøger aspekter af DSL *implementation* med et fokus på effektivitet og sikkerhed. I kontekst af MAL viser vi, hvordan vi implementerede en kodegenerator, som producerer kode, der er mere effektiv end sammenlignelig håndskreven kode mens der gives garanti om initialisering og bevaring af reserve.

Herefter kortlægger vi etablerede praksisser i DSL'ers *livscyklus* gennem en spørgeskemaundersøgelse af skabere af historisk vigtige DSL'er. I undersøgelsen fokuserer vi på forskellige faser i livscyklussen og viser blandt andet at, (a) vi ikke finder en korrelation mellem niveauet af brugeres inddragelse i designprocessen og brugeres programmeringserfaring, og (b) at de fleste DSL'er udvikler sig, og at hvordan denne udvikling håndteres påvirker et DSLs succes.

Endelig beskriver vi igangværende arbejde indenfor såkaldte Dynamic Condition Response (DCR) grafer som modeller arbejdsprocesser udviklet af Hildebrandt og Mukkamala. Her præsenterer vi arbejdet henimod at give hemmelighedsgarantier når parter med forskellige mål indgår i samme arbejdsproces.

Acknowledgement

Thanks to Peter Sestoft for supervising me during my bachelor's thesis, master's thesis, and now my PhD thesis. Throughout this time, Peter has provided me with support, guidance, and challenges. He has remained calm when I have encountered problems and shared his both wide and very specific experiences.

Thanks to people at the Actulus department in Edlund and, in particular, Henning Niss for endlessly supporting me, fighting on my behalf, and challenging my ideas.

Thanks to Christoph Seidl for helping me focus my thesis and research, sharpen my ideas, navigate academia, and improve my writing.

Thanks to Ulf Norell for hosting me at Chalmers University of Technology and giving me a glimpse into the universes of dependant types.

Thanks to Morten Tychsen Clausen for diligently helping me improve and implement the Management Action Language, and to Maria Bendix Mikkelsen, Christian Myrup Albinus, Simon Stampe Leiszner, Mikkel Rahlff Berggreen for your work and investigations related to MAL.

Thanks to Frederik Madsen for your friendship, support and collaboration during our bachelor's, master's, and PhD studies.

Thanks to Søren Debois for treating me as an equal from early in my bachelor's studies.

Thanks to all past and present colleagues in SQUARE for academic sparring and social events.

Thanks to Innovation Fund Denmark for funding the project (7076-00029B).

Thanks to all my family and friends and to Stefan Borum and Tilde Nor Stadel Borum for proofreading and editing.

Thanks to Sidsel Engmann Juul for never-ending love and support.

Revision

This version of the thesis have been revised based on the feedback in the *Prelimenary Report on the Phd Thesis*.

Contents

Chapter 1

Introduction

This PhD project was part of the research project *Projection of Balances and Benefits in Life Insurance* (Probabli), which investigates how to perform solvency calculations for Danish pension companies from an actuarial and a software engineering perspective. Three parties have participated in the project: the University of Copenhagen researched the actuarial mathematics of balance (consisting of assets and liabilities) projections, we at IT University of Copenhagen investigated balance projections through software language engineering, and the software company Edlund A/S (Edlund) provided industrial expertise on both the mathematical and software engineering perspective with its new balance projection platform.

1.1 Context of Thesis

The Danish pension sector is unusually large compared to the size of the country's economy. The sector manages reserves equivalent to 2 times Danish gross domestic product [1] and is thus of societal significance. The Probabli project exemplifies how the responsibilities of a Danish pension actuary have changed within the last couple of decades. In addition to their traditional area of work, actuaries now also often play a role as both users and developers of complex and high-impact software projects. This development has been caused by the availability and low cost of computational resources that have made it feasible to run large-scale balance projections of a pension company's portfolio of pension insurance policies. A pension company uses balance projections to calculate or approximate different quantities of interest, such as reserves and payment streams. The company's internal motivation for making these projections originates from an interplay between the company's motivation for improving its business, on the one hand, and new external financial regulations, on the other hand. Still, the balance projections require substantial software engineering efforts from pension actuaries.

This thesis presents and contextualises our work with creating the domain-specific language (DSL) called Management Action Language (MAL). We previously the ongoing work with MAL in the master's thesis [2]. A DSL is a programming language designed to be well suited for solving particular, delimited kinds of tasks. As a form of software product line [3], MAL's purpose is to easily let different Danish pension companies use the same balance projection platform even though they are managed differently. We present the work in the thesis Chapter 2 through 4, focusing respectively on MAL itself, non-technical DSL design, and DSL implementation. MAL is the primary source of empirical data augmented with experiences from related projects.

We designed MAL to solve a narrow but substantial problem primarily relevant to Danish pension companies, but we see its creation within the broader context of actuaries undertaking new roles. In other words, we created MAL in response to the actuarial field evolving, and such evolution is not unique to the field of actuaries. On the contrary, the availability of computational resources has caused many non-programming professionals to take on similar roles as users and developers of complex software products. We use the term *non-programming professionals* to keep the thesis consistent with the included papers, although Shaw makes a strong argument for using the term *vernacular software developers* [4]. Therefore, while the thesis takes the specific perspective of our experiences working with MAL within the actuarial domain, we generalise to the common case of non-programming professionals when possible.

1.2 Chapters and Research Contributions

DSL research is a diverse field concerned with the engineering process, from designing to creating to eventually retiring a DSL. The research involves methods, techniques, and tools supporting this process, along with many experiences and examples of applying DSLs as solutions to problems in different domains and contexts [5]. The research focused on supporting DSL development has primarily revolved around techniques for doing so with less focus on tools, evaluation, and integration with software engineering processes [6]. Furthermore, this research has overwhelmingly considered domain analysis, design, and implementation with almost no attention to maintenance and validation of DSLs [6].

The focus on design, domain analysis, and implementation is largely shared by our papers (included in Appendices A-E) that primarily present our work related to MAL (Table 1.1). However, the thesis also investigates the broader life cycle of DSLs, which includes maintenance and evolution and other life phases such as

launch and retirement. The main contributions are in the papers (Appendices A-E). Chapters 2-6 give an overview that contextualises the papers and additionally updates and reflects the papers' content when relevant. The thesis is structured as follows:

In Chapter 2, we give a brief introduction to MAL by walking through the implementation of a small module calculating the tax on yields from pension scheme assets, and we compare it to the existing solution. Furthermore, we describe some of MAL's language guarantees.

In Chapter 3, we summarise our contributions to the design of DSLs from both a human-centred and a co-design perspective. Furthermore, we argue for classifying DSLs according to their problem or challenge space instead of their solution space, domain, or implementation technology. We argue for the utility of such a classification by showing how the challenge space of MAL affected its design process compared to similar DSLs.

In Chapter 4, we describe the current state of MAL's language implementation in terms of IDE support, code-generator, and MAL template code. We describe the work leading to a code generator producing code that is between 1.27× and 1.46× faster than comparable handwritten C#. Furthermore, we discuss and analyse why it is difficult for MAL to move from a reserch project to an Edlund product.

In Chapter 5, we present the findings of our survey of established practices in the life cycles of DSLs appearing in different curated DSL collections. We discuss the findings relating to user involvement in DSL design, evolution, and pragmatism in relation to our other work presented in the previous chapters.

In Chapter 6, we present ongoing work into statically providing secrecy guarantees for Dynamic Condition Response graphs that model distributed workflow processes. We first informally argue for the need for an approximation of secrecy and then sketch an approach to approximation inspired by information flow analysis.

In Chapter 7, we conclude the thesis by summarising the thesis' contribution and discussing how two underlying themes of evolution and pragmatism appear in the different thesis chapters. Furthermore, we present five avenues of future work relating to DSLs, user evaluation, and workflow secrecy.

This thesis makes the following new contributions and updates to our articles:

- A comparison between a specification written in MAL and the specification

Table 1.1: The constituent papers of the thesis, their status, a summary of their contribution, and which thesis chapter they primarily relate to.

App.	Ch.	Title	Status	Venue
A	3	On Designing Applied DSLs for Non-Programming Experts in Evolving Domains	Published	MODELS'21
A classification of languages according to evolutionary characteristics, a proposed DSL design method, and a case study of MAL's design process.				
B	3	Co-designing DSL Quality Assurance Measures for and with Non-programming Experts	Published	DSM'21
A presentation of MAL, an approach to and experiences with co-design within DSLs, and the design of debugging spreadsheets as a general quality assurance measure.				
C	4	Transforming Domain Models to Efficient C# for the Pension Industry	Submitted	MLE'22
An identification of common operations in management action specifications, a strategy for generating code for these specifications, and benchmark results.				
D	5	Survey of Established Practices in the Life Cycle of Domain-Specific Languages	Accepted	MODELS'22
A presentation of empirical data regarding DSL's lifecycle, an analysis of this data, and empirically based recommendations.				
E	6	Static Secrecy Guarantees for Dynamic Condition Response Graphs	Working Paper	(TBD)
A formalisation of secrecy approximation in DCR executions, our work towards an efficient secrecy approximation, and the concept of minimal runs.				

as handwritten management template code. (Section 2.2)

- A proposal and draft for creating a problem space DSL typology. (Section 3.3)
- An analysis and discussion of problems MAL is facing in transitioning from a research project to a product. (Section 4.4)
- An informal presentation of our *information flow* inspired approach to secrecy analysis in DCR graphs. (Chapter 6)

Chapter 2

Management Action Language

The Management Action Language (MAL) was designed in an industrial collaboration between Edlund and the IT University of Copenhagen to improve Edlund's balance projection platform. MAL is a spiritual successor of the domain-specific language Actulus Modeling Language (AML) [7], which was designed and developed in a similar collaboration between the same institutions. Whereas AML allows actuaries to specify pension products, MAL allows actuaries to specify how pension product instances are managed in balance projections. The two languages are only integrated to the degree that input values to a balance projection may originate from an AML specification. In this chapter, we present MAL as follows: First, we give a brief motivation for creating MAL by describing the system it is designed to replace. Second, we present MAL's concrete syntax through a small module definition and compare it with the existing solution. Third, we present important language guarantees provided by MAL. A snapshot of MAL's typechecker and code generator is public available [8], but the entire code base is not publicly available as it integrates with proprietary DLLs.

2.1 Language Motivation

Edlund is a Danish software company specialising in creating software for the Danish pension industry. One of Edlund's products is a platform that allows a pension company to perform balance projections in accordance with financial regulations. The projection platform (Figure 2.1) is separated into calculations that are the same for all pension companies (the *projection engine*) and calculations that are specific to a given company (so-called *management actions*).

The purpose of MAL is to allow actuaries in pension companies to express efficient, company-specific management action in a secure and correct way. From the perspective of software product lines [3], MAL is a way to facilitate software

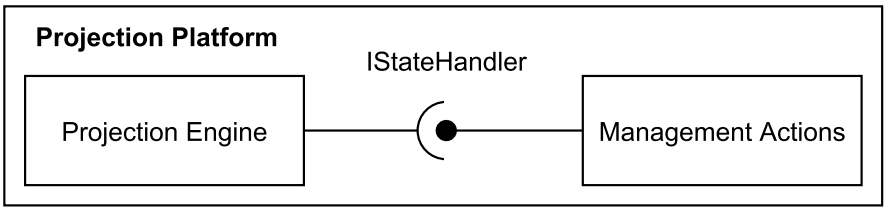

Figure 2.1: A simplified depiction of the projection platform consisting of Edlund's general projection engine and the company-specific management actions.

variability by allowing a pension company to specialise the projection platform with the company's business rules. Currently, Edlund allows companies to submit management actions described by implementing a C# interface to the projection platform. To help customers, Edlund has created a sizeable, modifiable template solution in C#. We designed MAL to replace this solution and allow for the creation of a similar template solution and easier customisation of the template.

2.2 Language Example

We present MAL by specifying a module that calculates tax on yields from pension scheme assets (PAL, short for *pensionsafkastbeskatningsloven* [9]). The module computes the tax for pension *policies* and *groups*. In Danish pension tradition, a group is a collection of pension policies that share specific dividend payments. We compare the specification to equivalent real-world template specification in C#. For brevity of presentation, we use a simplified data model and do not show initialisation, but we remain true to how the template specifications look.

The `PalValues` module is a small computational unit that is a fifth of the size of the largest one, measured in lines of code. We start by defining a skeleton with C++-style comments as numbered holes that we will fill in later:

```
module PalValues
// ① Data contracts
// ② Action and Function declarations

export action manage()
{
   // ③ Management
}
```

The above declaration defines the module `PalValues`, which exports a `manage` action that takes no arguments. We will proceed by defining: ①, the module's data contracts that specify what data the module requires and provides. ②, the module's local actions and functions used in the calculations. ③, the computational content of the `manage` action.

In C#, a class is used for a similar computational unit. It consists of: ①, a number of read-only fields that specify the result of the calculation. These values are returned to the caller of the static `Calculate` method that instantiates a `PalValues` object. ②, utility methods used in the calculations. ③, the body of the Calculate method that contains the computational content of the unit.

```
class PalValues
{
   // ① Read-only fields
   // ② Method declaration

   static PalValues Calculate(
     PalValues previousPalValues,
     IReadOnlyDictionary<string, PolicyPeriodResult>
       policyIdPeriodResult,
     IReadOnlyDictionary<string, GroupPeriodResult>
       groupIdPeriodResult)
   {
      // ③ Management
   }
}
```

① Data contracts in MAL

The PAL template solution calculates the tax on yields from pension scheme assets for both pension policies and groups.

```
export data PalTax
   InvestmentReturnTaxAsset : Float
   InvestmentReturnTaxPaymentForPeriod : Float
end
```

The above code defines a data entity for `PalTax` that contains two floating-point numbers specifying the investment return asset and payment for the period. We use the definition to specify that the module calculates `PalTax` for both policies and reserve groups with the following data contracts:

```
contract Policy
{
   requires {
      Result : PolicyResult
   }
   provides{
      Pal : PalTax
   }
}
contract ReserveGroup extends Group
{
   requires {
      Result : GroupResult
   }
   provides {
      Pal : PalTax
   }
}
contract Global
{
   requires {
      PalTaxRate : Float
   }
}
```

The above code specifies that the `PalValues` module uses the `Result` of both a `Policy` and a `ReserveGroup` to calculate their `PalTax`. Also, the `Global` object must have a `PalTaxRate`. We omit some empty data contracts required by the type checker.

In C#, class fields describe the values provided by a computational unit:

```
Dictionary<string, double>
   PolicyIdInvestmentReturnTaxPaymentForPeriod,
   PolicyIdInvestmentReturnTaxAsset,
   GroupIdInvestmentReturnTaxPaymentForPeriod
   GroupIdInvestmentReturnTaxAsset;
double PalTaxRate;
```

There are three important aspects of the C# declaration to discuss. First, the specification follows a consistently used convention of storing computed values in a dictionary mapping identifier strings to floating-point numbers. While one could have created a class similar to `PalTax` to only manage one dictionary for policies and one for groups, this choice would have broken the convention. Second, the code above shows only values provided by the computations. The required values are most often specified as parameters to the `Calculate` method. While the pattern,

in this case, allows for a short specification, the size of a method's signature tends to become large and difficult to read when additional data is required due to many dictionary arguments. Third, it is unclear from the specification that `PalTax` is calculated only for a `ReserveGroup`. For our purpose, it suffices to know that a `ReserveGroup` is conceptually a subtype of a `Group`. Such a constraint that appears in comment documentation is easy to overlook, misunderstand, and mistrust.

② Action and function declaration in MAL

We now define how `PalTax` is updated by the `updateSinglePal` management action. A data field may be updated in a MAL action, whereas it may not in a MAL function.

```
action updateSinglePal( Pal : PalTax
                      , realisedReturn : Float)
{
   let investmentReturnTaxForPeriod =
        realisedReturn * Global.PalTaxRate
   let startPeriodAssets =
        Pal.InvestmentReturnTaxAsset
   let investmentReturnTaxPaymentForPeriod =
        max(investmentReturnTaxForPeriod - startPeriodAssets, 0)
   let investmentReturnTaxAsset =
        startPeriodAssets
      - investmentReturnTaxForPeriod
      + investmentReturnTaxPaymentForPeriod
   Pal.InvestmentReturnTaxPaymentForPeriod =
        investmentReturnTaxPaymentForPeriod
      - investmentReturnTaxAsset
   Pal.InvestmentReturnTaxAsset = investmentReturnTaxAsset
}
```

The action primarily consists of simple arithmetic and assignments that we will not describe in detail. It is worth noting that the action makes use of the globally available entity `Global`.

In C#, a method is used for a similar computational definition. We do not show the content of the method as it is verbose and difficult to format in a readable manner. The method is implemented by iterating through the dictionaries in the argument of the method.

```
static Tuple< Dictionary<string, double>
            , Dictionary<string, double>
            > CalculateInvestmentReturnTaxAssets
   ( double palRate
   , IReadOnlyDictionary<string, double> investmentReturnTaxAsset
   , IReadOnlyDictionary<string, double> realisedReturn
```

```
   )
{
   // Omitted 14 lines and 639 characters formatted for an IDE
   // In comparison the entire above MAL action has 13 lines
   // and 630 characters when formatted for an IDE
   return Tuple.Create( investmentReturnTaxPaymentForPeriod
                      , investmentReturnTaxAsset);
}
```

③ Management in MAL

Finally, we can specify the behaviour of the module's manage action as follows:

```
update policy in Policies
{
   do updateSinglePal( policy.Pal
                     , policy.Result.PeriodRealisedReturn)
}
update group in Groups:ReserveGroup
{
   do updateSinglePal( group.Pal
                     , group.Result.PeriodRealisedReturn)
}
```

The above code calls the previously defined action `updateSinglePal` to compute the `PalTax` for both policies and reserve groups. Note the usage of the type filter construct `Groups:ReserveGroup` that chooses all reserve groups from the collection `Groups`.

In C#, the same calculations are made by similar calls to the previously defined method `CalculateInvestmentReturnTaxAssets`. However, before a call, argument data must be manipulated to be compatible with the method's signature, and after both calls, data is manipulated to return a `PalValues` object.

```
var policyIdPeriodRealisedReturn = policyIdPeriodResult.Map(
  (policyId, resultForPolicyId)
    => resultForPolicyId.PeriodRealisedReturn);
var policyIdPalResults =
   CalculateInvestmentReturnTaxAssets
     ( previousPalValues.PalTaxRate
     , previousPalValues.PolicyIdInvestmentReturnTaxAsset
     , policyIdPeriodRealisedReturn);
//... groupIdPalResults is calculated in a similar manner ...
return new PalValues
   ( policyIdPalResults.Item1
   , policyIdPalResults.Item2
   , groupIdPalResults.Item1
```

```
  , groupIdPalResults.Item2
  )
```

In this code fragment, we see some of the consequences of storing computed quantities in dictionaries. First, a considerable amount of dictionary manipulation has to be performed to match method signatures. Second, many intermediate dictionaries are created only for short-term use. An example of this is `policyIdPeriodRealisedReturn`. While it is easy to point to problems with using dictionaries in this fashion, it should be noted that the template developers have reasons for doing so. First, the dictionaries are versatile in that it is easy to create a quantity locally without having to worry about how it would fit in an inheritance hierarchy. This property of locality is especially relevant in that it makes it easy for customers to introduce local modifications. Second, the dictionaries are compatible with the projection engine's interface. Another choice of data structure would necessitate a significant amount of conversions to be compatible with the interface.

Summarising the MAL vs. C# comparison, we have shown how we, in the design of MAL, have included and improved concepts appearing in the template specification. Instead of using classes to encapsulate a computational unit, MAL uses modules with explicit data contracts declaring what values the module requires and provides. Instead of using local string-indexed to store computed values, MAL expands data entities with new fields when needed. Finally, we have not shown how to output values from a projection. However, it should be mentioned that this can be done in a declarative manner in MAL by annotating a data declaration.

2.3 Language Guarantees

We have already hinted at some usability and guarantees provided by MAL. Still, we want to treat important language guarantees in their own section. We use the word *guarantee* in its informal meaning of property provided by the language without a formal proof of soundness. We previously used Agda [10] to prove a subset of MAL [2] to be type-safe but found the time cost of creating proofs to hinder the development of MAL.

2.3.1 Initialisation Before Use

MAL uses explicit data models and data contracts to statically guarantee that a field is initialised before it is used. This guarantee comes in two flavours: First, within a module, all fields are assigned before they are used. Second, before calling into a module, all of the module's required data fields must be initialised. This guarantee should be seen in contrast to the usage of identifier-index dictionaries to

keep track of values in the C# solution. In the above example, a string-indexed dictionary is used to keep track of the `InvestmentReturnTaxAsset` of policies and groups. Here it is not clear that such a value exists for all policies but only for specific types of groups. Furthermore, there is no guarantee that a value exists at a specific identifier in a dictionary, i.e., there is no guarantee that the value has been initialised. Although we have not seen the need, MAL's initialise-before-use check could be upgraded to an update-before-use check that ensures a computation only uses values updated within its time period.

2.3.2 Relationship Cardinality

Management actions traverse a relationship graph between policies and different types of groups. MAL ensures the cardinality of different kinds of relations. For example, all policies must belong to exactly one *interest* group but may belong to zero or one *expense* group. Again, this guarantee should be seen in contrast to the C# solution where, as part of the interface, these relations are stored in identifier-index dictionaries with no cardinality guarantees.

2.3.3 Reserve Preservation

MAL is to be used to manage large monetary reserves. In doing so, MAL ensures that the sum of reserves remains constant so that it is impossible to invent or lose money by accident. In relation to this property, a MAL program is correct by construction in that there is an explicit construct for reserve transfer. An example of the construct is seen below that transfer 200 from the reserve of `group` to the reserve of `policy`.

```
group.Reserve |> 200 |> policy.Reserve
```

2.4 Summary

In this chapter, we have presented MAL, which we designed for actuaries to express management actions in balance projections. We have done so by defining a module for PalValues and comparing it to existing handwritten C#. First, we defined the data contract of the module. Second, we defined an action for updating the PalValue of a single policy or group. Third, we defined the manage action that iterated through all relevant policies and groups. In creating this definition, we have seen how MAL lets users modify and extend the data model in a manner where the language implementation guarantees that values are initialised. In addition,

MAL's explicit data model makes it possible to provide guarantees on relationship cardinality, which is difficult to do in the current C# setup. While we in this chapter have argued for the utility of MAL programs in terms of code structure, readability, and error-proneness, MAL also generates efficient code, as we argue in Section 4.3.

Chapter 3

DSL Design

Domain-specific languages promise their users support tailored to their tasks. Nevertheless, many DSL users still have to engage with a complex, abstract software system to solve programming tasks with a multitude of solutions. The purpose of DSL design is to help users in this task by making it easier to create large, complex, secure, efficient, and correct software solutions. Therefore, a DSL designer should get methodological prescriptions for designing a usable DSL. We contribute to the investigation of how human-centred design (HCD) can be used for such methodological prescriptions through our experiences with creating MAL. The important HCD activity of *evaluation* has, arguably, been neglected as a DSL design activity [11]. Several practitioners propose early user evaluation of DSLs [12, 13]. However, such evaluation cannot simply be introduced in a development process but must be facilitated by other activities. In this chapter, we first present the field of DSL design from the perspective of HCD, then we summarise and discuss our research contributions, and finally, we discuss the utility of a problem-space oriented classification of DSLs as opposed to the current solution space classification.

3.1 Human-Centred Design

Human-centred design seeks to include the users in the design process to ensure that the designed artefact is usable [14]. We use the terms *user-centred design* and *human-centred design* interchangeably, although the latter may be argued to have a broader socio-technical design perspective [15] similar to *participatory design* [16]. For such a design process, we prefer to simply use the term *participatory design*. While Myers et al. [17] and Coblenz et al. [18] present HCD techniques for programming language design, we instead present example applications of design techniques during the different design activities. Besides a planning activity (which we return to in Sections 3.2.1 and 3.3), human-centred design is an iterative process consisting of the four activities [19] presented below.

Context-of-use elicitation investigates the setting in which the DSL is to be used. It is difficult to find reports on conducting this activity, but a prominent example is a usability study of a similar software product [20]. More often, papers present the context-of-use as a given prior for a design project with no methodological description, e.g., [21, 22, 23]. We conjecture that there are three reasons for this scarcity of reports on context-of-use elicitation. First, context-of-use elicitation of DSL projects may not differ from the general case of software engineering. Second, the problem tackled by DSL design may have already been established as being important. Third, the authors of papers regarding DSLs do not see context-of-use elicitation as a primary central contribution of their work.

Requirement elicitation establishes the functionality that must be available in a DSL. Requirements are often stated with no methodological description as to how they were found. We find that requirements often come from a) the system that the designed DSL is to replace [24], b) the analysis of established domains [25], or c) case study analysis [26]. Again, we conjecture that the scarcity of methodological considerations are due to them not being perceived a primary challenge by DSL practitioners.

Producing design solutions creates or prototypes the DSL for subsequent evaluation. For this activity, we find it useful to distinguish between *idea generation* methods and *prototyping* methods.

For *idea generation*, we believe the dominant methods to be bottom-up analysis, brainstorming, and rapid prototyping, but we again find few reports from practitioners applying such methods. A frequent piece of advice is to "[a]dopt whatever formal notations the domain experts already have, [...]" [27, 28] which suggests a bottom-up analysis. *Natural programming* [29] is one generative method that has been applied to adopt the notation and conceptual model of users [20, 30]. A similar approach is adopted by different versions of example-driven meta-model development [31, 32, 33, 34] for domain-specific modelling languages (DSMLs). Zaytsev created the framework *Language Design with Intent* [35] that presents 96 actionable design decision points for software language designers.

For *prototyping*, one needs to create an artefact for both design-space exploration and evaluation. Language workbenches [36, 37] help in the implementation of DSLs. A recent systematic mapping study finds that the most used tools are XText, Eclipse Modeling Framework (EMF), and MetaEdit+ [38]. However, a prototype does not need to be a fully functioning language design since non-functioning mock-ups may

still allow for exploration and evaluation. Whiteboard and magnets [26] can be used for collaborative modelling. Language backporting [39] implements design experiments in a mainstream host language. Different *Wizard of Oz* [40] techniques are used to mimic systems that have been only partially implemented [41, 39].

Evaluation seeks to find strengths and flaws in a DSL with the purpose of refining it. For HCD, it is important that users are involved in this evaluation, which means that methods that do not involve users, such as cognitive walkthroughs, heuristic evaluation, or other forms of expert evaluation, are only supplementary. Also, we recognise the difference between scientific evaluation and evaluation for the purpose of design validation. While there is an overlap between the two activities, our interest here is the latter. A recent, updated systematic literature review [42, 43] finds usability evaluation combined with recordings or questionnaires to be the predominant HCD method. Barišić et al. [11] present a conceptual framework for adding an evaluation as a phase in DSL development. Like us, they emphasise the need for early user evaluation but assume a design context with many available users. Evaluation with different users at different stages may be used to progressively ensure different usability goals [13]. Collective task solving conducted at workshops can be used to refine a DSL [44]. Evaluation may be an open house activity where guests willingly participate in different experiments [45]. Other evaluation has used randomised controlled trials for in-depth examination of different topics, such as the impact of choice of keywords [46] or the introduction of lambdas in C++ [47].

3.2 Discussion of Research Contributions

In our paper *On Designing Applied DSLs for Non-Programming Experts in Evolving Domains* [48](Appendix A), we explored HCD of DSLs through the design and implementation of our domain-specific language MAL. We propose a two-phase design method for designing DSLs in evolving domains. The method moves from a phase of *low-certainty exploration* using in-team domain experts for lightweight evaluation to a phase of higher validity *design validation* using external domain experts. A more participatory design approach was investigated in our paper *Co-designing DSL quality assurance measures for and with non-programming experts* [49](Appendix B) where DSL users were generatively engaged in the idea generation activity. In this section, we critically discuss selected contributions of these papers.

3.2.1 Challenge Identification

We synthesised the proposed two-phase design method during the design of MAL. This process means we generalise our concrete experiences to a wider set of design contexts where we expect them to be relevant. We do not think that we can generalise by ignoring specific challenges. On the contrary, the investigation of specific design challenges is the foundation for the generalisation. As an example, we identify the challenge that user involvement in the design process has to be limited due to the few available users and the scarcity of their time. This challenge led to the concrete advice that during the first design phase, only an in-team domain expert should be used for evaluation. In a design context where many users who can devote time to evaluation are easily available, the prescription would likely be different.

The described method for generalisation implicitly bounds the applicability of the proposed design method to those that have similar challenges to ours. From this perspective, it is natural to consider whether our design experiences could be more generally applicable by making *challenge identification* an explicit step of the method similar to the planning activity in HCD [19]. However, in doing so, the proposed method could suffer from a lack of prescriptions since our experiences do not allow us to prescribe what to do in other design contexts. Thus, any prescriptive method has to strike a balance between the overly general advice of "Do something reasonable" to specific advice of the form "Here is something reasonable that we did". We return to the subject of challenge identification in our discussion of how a problem space DSL typology may be useful for DSL practitioners (Section 3.3).

3.2.2 Evolution and Requirement Elicitation

Language evolution is an aspect of DSL creation that played a significant role in the creation of MAL [48](Appendix A). Amstel et al. [50] found that their DSL evolved in response to changes in its *problem domain* and *application domain* and to improve the *quality of models* and the *quality of transformations*. These causes of evolution adequately characterise our experiences with creating MAL, but we find it useful to further analyse *problem domain* evolution.

We found two kinds of evolution in MAL's problem domain. First, there was a *problem-domain expansion* that introduced new concepts and rules to be expressed or changed in existing concepts. Second, there was a *problem-domain settlement* when solutions to previously unsolved problems appeared. The latter kind of evolution is a symptom of what we call an *amorphous domain*, i.e., a domain with many uncertainties. As an example, MAL's problem domain of balance projections is defined by the following four stakeholders. First, the Danish Financial Supervisory

Authority implements the rules within the framework of EU legislation. Second, Danish pension actuaries develop different mathematical models of how to perform balance projections. Third, each Danish pension company has to individually decide how to best implement financial regulations for its business. Fourth, software vendors and developers have to figure out what is technically feasible. The combination of these stakeholders and a young domain means that there have been many unknowns in MAL's problem domain while it has slowly settled. Even in 2022, there still are unsolved problems; most stakeholders are seemingly perplexed when considering how to allow decisions to be made on prospective values, if doing so is even possible.

An amorphous domain is problematic for DSL design since it makes requirement elicitation an uncertain and ill-defined task. Domain experts may not be comfortable with delimiting the solution space since they see what we may call *potential solutions* to a given problem. Although it may be easy for a designer to create constructs for all potential solutions, doing so will tend to make the language more *general-purpose* and contain superfluous constructs and features. We found that only designing the language for the currently established domain mitigated this problem to some degree. However, the realisation of potential solutions, such as adding customisable projection times or using a probabilistic data model, still caused language evolution. Still, we also saw requests for features that were not considered as needed for potential solutions but could easily have been handled by making the language more general-purpose, e.g., by adding traditional while loops.

The experiences with DSL evolution led us to a deductive categorisation of programming languages inspired by Lehman's *S-type*, *P-type*, and *E-type* programs. Our categorisation categorises according to their evolutionary characteristics but does not use Amstel's above-mentioned causes for evolution since they are not applicable to all types of language. We hypothesise that DSLs created as a software engineering solution are E-type programming languages. Like an E-type *program*, an E-type *language* is highly susceptible to evolution since it becomes part of and affects its own application domain. Although we have not investigated the categorisation inductively, we find other DSL practitioners have had similar ideas. Karaila [51] analysis 20 years of evolution of the Functional Block Language from the perspective of being an *E-type program* and finds that Lehman's laws [52] characterise the evolution well.

3.2.3 Prototyping, Evaluation, and Idea Generation

A user of a DSL seeks to find one of several solutions with different merits to an open-ended problem. They do so by expressing a plan to be executed in the future

while identifying and correcting errors based on a wealth of information [53]. Such interaction has fundamental differences from a well-defined task such as withdrawing money from an ATM and therefore questions the applicability of conducting traditional short-term usability tests [54]. From this perspective, we investigate prototyping and evaluation of DSLs [48](Appendix A) and how to engage users actively in idea generation [49](Appendix B).

To have a purpose, evaluation should have the possibility of affecting the DSL design. Therefore, we propose that designers should create low-cost and early prototypes with a corresponding lightweight evaluation. Such low-cost evaluation is especially important for novice designers who explore a domain new to them since it allows them to make and fix mistakes. We found during the phase of *low-certainty exploration* that creating non-functional *text-editor prototypes* allowed us to evaluate a diverse set of language designs early. We experienced that it is unreasonable to expect a domain expert to have opinions on different language constructs but that comparing and modifying solutions prompted feedback. Once the design settles, we propose validating the design with actual users on a functional prototype to mitigate the risk of having received biased feedback from the in-team expert.

Co-design is another approach to ensuring that a designed artefact is usable by its intended end-users. The idea is to engage users actively in the design process so that they can create solutions to their needs. MAL has certain compile-time guarantees (Section 2.3) and a prototype debugger implementation. Still, it was difficult for us to figure out what quality assurance measures domain experts needed when working with balance projections. Therefore we set out to co-design quality assurance measures in a workshop with prospective MAL users. Here we found that our prospective users approached quality assurance from the analytical perspective of understanding their solution, whereas we approached quality assurance from a testing perspective. Upon realising this difference, the workshop became fruitful, leading to three concrete designs for quality assurance. From these experiences, we conclude that DSL users can be engaged actively in creating designs, although bridging a gap in perspectives is a significant part of doing so.

3.2.4 What About Tools?

While DSL tooling is an important aspect of DSL research, the presented research contributions to human-centred DSL design have appeared as orthogonal to tooling. Here we discuss different claims of how different tooling might have affected our findings.

Language workbenches facilitate rapid prototyping
We found it important to create low-cost DSL prototypes for early design evaluation and therefore prescribe using text-editor prototypes. However, one may argue that since workbenches reduce the turnaround time of defining and implementing DSLs, they can and should be used for prototyping purposes. The argument would be that workbench-supported prototypes allow for more realistic evaluation since they provide a realistic programming environment. Still, we found a need to explore markedly different non-trivial designs and implementing each in a workbench would require significantly more implementation time than sketching them in a text editor. Also, having the DSL designer play the role of development environment was adequate for evaluation. So while it is possible that prototype workbench implementation is advisable in some situations (discussed in Section 3.3.3), we found the overhead too large when working with a complex DSL.

Language workbenches reduce the cost of evolution
We found language evolution to be a significant challenge in the development of MAL. One could hope that tool support would reduce some of this cost by, for example, facilitating co-evolving the language definition with its programs. However, we found that most of the time spent on language evolution was spent analysing evolved GPL programs and understanding this evolution's impact on the language definition. As a side note, the motivation for creating MAL, to a large extent, originated from the difficulty of understanding the GPL program. We spent relatively little time actually adjusting the language definition and program instances where tool support could practically have helped us.

3.3 DSL Typology

So far, we have purposefully avoided giving a strict definition of the term domain-specific language. The term is broadly used to encompass languages ranging from advanced configuration file formats to Turing complete languages. From the perspective of DSL design, we find the broadness of the term problematic since vastly different design approaches may be suitable for different types of languages. Tomassetti and Zaytsev [55] similarly find the broadness of the term an obstacle for DSLs being adopted in software engineering and suggest a subcategorisation similar to ours as a solution. In our context, the problem with the broadness relates to the planning activity in HCD that plans how usability activities are to be used during the project by exploring the wider design context.

Traditionally domain-specific languages are presented using solution space categorisations. That is the dichotomy of *internal* and *external* DSLs, the dichotomy

of *interpretation* and *code generation*, and sometimes the differences between projectional editing and parsing or textual and visual languages. While knowing the solution space characteristic of a DSL guides a creator towards its technical design, it does not provide a starting point for the language design process. Here we present a starting point for a problem space typology of DSLs. We argue for the utility of the individual dimensions by hypothesising guidelines for the design process. We only hypothesise since our limited experience with working with MAL does not allow us to make stronger claims. Still, we find the guidelines reasonable and think they or others should be investigated empirically. Also, we see a similar idea in the context modelling activity of the *USE-ME* framework [11] that determines DSL evaluation from the problem space. Zaytsev shares our interest in having design guidelines with the aforementioned *Language Design with Intent* toolkit. However, we find the framework to be of too fine granularity and too solution-oriented to be actionable. Similarly, Poltronieri et al. propose the evaluation framework *Usa-DSL* [56] to support designers in the evaluation of DSLs. Although the framework presents many used evaluation methods, metrics, instruments, and so forth, it provides little to no guidance on which methods to use in a given situation.

3.3.1 Users

The target users of different DSLs vary in programming experience, availability, group size, and level of heterogeneity. All of these aspects influence what level and kind of evaluation is possible and needed in a design project. Halvorsrud et al. found that the heterogeneity of their users made its usability evaluation challenging [44]. Therefore, they made a large-scale workshop evaluation which was possible due to the availability of users. We found usability testing to be important due to users' limited programming experience and sought to make the best use of their time due to their limited availability [48](Appendix A). We conjecture that when target users are a homogenous group of programming experts, then usability testing becomes less important due to their mental model being close to the designer's. Still, if the designed DSL introduces advanced or esoteric programming concepts or notation, the design should be validated.

3.3.2 Tasks

The tasks carried out by users of different DSLs vary in their open-endedness and the kind of their task (read, modify, or write DSL code). We conjecture that for open-ended tasks, usability evaluation requires longer sessions with users, whereas heuristic evaluation or cognitive walkthroughs may be adequate for tasks with a single solution. Also, whether users are to primarily read, modify, or write DSL

fragments should be reflected in the usability evaluation and potentially even in the design of the DSL.

3.3.3 Evolution Characteristics and Application Context

The evolutionary characteristics and application context of a DSL influence when and how to implement or create prototypes. We have already discussed different kinds of evolution in Section 3.2.2, along with a brief presentation of our evolution-based characterisation of DSLs. While our characterisation could be used in this context, it is more important to identify how likely the DSL is to evolve and why. Connected to this question is identifying the application context of the DSL, for instance, whether the DSL is to serve as a thin interface to an existing library or to be compatible with a complex existing infrastructure. We conjecture that early DSL implementation is suitable for stable domains with a simple existing application context, whereas prototyping becomes important for more evolving domains and complex application contexts.

3.3.4 Development Context

A design project takes place in a development context that may dictate the order of some activities. While an iterative development process is often prescribed as suitable for a DSL project, DSL practitioners may find themselves having to conform to a linear development process or maybe even open source or collaborative development. We conjecture that one should seek some form of early user evaluation even in linear development, but if not possible, one should rely on expert evaluation. By contrast, in extreme cases of agile development, all user evaluations could come from actual DSL usage.

3.3.5 System Impact

When a DSL is developed as part of safety-critical systems, one has to consider how to ensure the DSL implementation and integration are correct [57]. We conjecture that usability evaluation of such a DSL is also important for the purpose of reducing the risk of user-made errors. Therefore, such a safety-critical DSL should have earlier functional implementation and more thorough evaluation activities with users. On the contrary, DSLs may also be developed for software systems where mistakes are benign or even interesting (e.g., languages for the creation of art). For such systems, usability evaluation may be less important or may have another focus than reducing user-made errors.

3.3.6 DSL genres

Different genres of domain-specific languages have been established to solve more general domain-specific problems [12, 58]. It seems meaningful to establish design methodology within these different genres if possible since these languages are often reminiscent of each other in some aspect. We do not propose such descriptions but find genres such as *task specification languages* [59], *financial product languages* [60, 61, 7], or *format description languages* [62, 63] relevant.

3.4 Summary

In this chapter, we have contextualised and discussed our work with human-centred design and co-design of DSLs. First, we have presented the four phases of HCD while presenting related work within each phase. Second, we have critically examined our contributions by discussing our findings in terms of their generalisability and whether they were affected by our choice of tooling. Finally, we have argued for the utility of a finer granularity categorisation of DSLs for improving guidelines for DSL design. Based on our experiences, we propose six problem-space dimensions that we conjecture influence the process of a design project. These influences come in two forms, a) what kind of process is required in a given design situation and b) what kind of process is possible in a given design situation. We think this kind of categorisation is in itself a small step towards providing DSL practitioners with design guidelines.

Chapter 4

Implementation of Management Action Language

In our survey of established practices in DSLs' life cycle [64](Appendix D), we find that DSLs are most often developed to improve users' productivity or code quality in terms of correctness and understandability. In this chapter, we first describe why MAL belongs to a minority of DSLs where program efficiency matters due to the high computational cost of performing balance projections. Second, we contextualise our work by presenting other approaches to balance projections in the Danish pension industry. Third, we give a high-level description of MAL's implementation and discuss the benefits of generating code for management actions. Finally, we discuss why MAL is unlikely to transition from a research project to an actual product despite our perceived benefits of using MAL.

4.1 Computational Cost of Projections

The purpose of a projection is to examine how the assets and liabilities of a pension company develop far into the future to ensure that the company remains solvent throughout this period. A projection typically involves moving a company's portfolio (app. 10,000 policies with 100-step cash flows) 100 years into the future, over 1,000 different economic scenarios. A projection must, for each economic scenario, step through all years, and for each year, iterate through all policies and make computations on its cash flows. There is a considerable computational cost in making such projections, and Danish pension companies consider it important to lower this cost. Danish financial regulations allow pension companies to have different fundamental methods for making these projections. MAL is specifically designed for a projection platform that essentially performs a Monte Carlo simulation. Although there is a diversity in methods, pension companies use the following two methods for improving the performance of projections:

Sacrifice Precision
The performance of projections may be improved at the cost of the precision of projections. From this perspective, many parameters may be changed more or less cleverly. To mention three examples: First, a company may select a few representative policies to use in a projection and thereby avoid the substantial computational cost of using its entire portfolio. Second, a company may choose to use fewer and longer time steps used in a projection. Third, a company may choose different underlying policy state models, such as a three-state or one-state model.

Efficiency of Specification
The performance of a projection may be improved by optimising different parts of a specification with no loss of precision. Such optimisations can either be in the form of choosing better procedures (e.g., solving partial differential equations analytically rather than numerically when possible) or optimising the procedure itself (e.g., improving memory allocation and layout, changing the order of operations, parallelising execution, and reusing computations when possible). The optimisations used by MAL's code generator all fall into this latter category.

4.2 Extent of the MAL Implementation

From the perspective of implementation, we regard MAL's implementation to be at a stage where it could be taken into production (apart from Edlund assuring the quality of MAL). The implementation consists of:

- A fault-tolerant parser.
- A static analyser (typechecker and initialisation check).
- A code-generator producing efficient C#.
- Visual Studio Code support through the Language Server Protocol [65].
- Template implementation in MAL corresponding to approximately 10,000 lines of template management actions in C#.
- Property-based testing [66] of a subset of MAL testing the parser, the typechecker, and the code generator.
- A documentation website used for usability testing.
- An early prototype of debugging support using the Debugger Adapter Protocol [67].

4.3 Generating C# Code for Management Actions

Since a code generator is a constituent part of DSL, a code generator is, along with its DSL, tailored to a domain and in itself tailored to a target language. This specificity means that while code generation is relevant to external DSLs, implementations are distinct for different languages. For example, Reiche et al. [68] generate high-level synthesis code from an image processing specification, and Barriga et al. [69] generate code for different hosts that simulates an internet of things system from a model specification. Still, general techniques for code generation, such as *deforestation* [70, 71] and *vectorisation* [72], are applicable to many code generation contexts. In our paper *Transforming Domain Models to Efficient C# for the Pension Industry* [73](Appendix C), we describe how we created MAL's code generator from identified common management-action-specification patterns and general code-generation techniques. Here we first describe other approaches to executing management actions, then discuss the implementation from the perspective of implementation, and, finally, discuss the benefits of using code generation in our project.

4.3.1 Executing Management Actions

We have, so far, primarily used the term *management action* to describe the business rules of pension companies as they appear in the projection platform. However, the term is general to at least all Danish pension companies that perform balance projections. In other words, Danish pension companies must perform balance projections based on management actions, and therefore these companies all[1] need an executable formalisation of their management actions, whether they use Edlund's platform or not. From observing advisory board meetings for the Probabli project and an industry conference, we find that Danish pension companies specify their management actions in general-purpose languages. This specification is either part of in-house balance projections or input to a projection platform, possibly a product of one of Edlund's competitors. To our knowledge, our project is the only DSL that lets actuaries express management actions.

[1]To be precise, it is possible that a company may not need to define management actions if it only sells products where the customer assumes all financial risks, and the company thereby assumes none.

4.3.2 Implementation Process

In [73](Appendix C), we present MAL's code-generator by stating that we identified pervasive management action patterns and then describing efficient code generation for these patterns leading to a significant speedup of execution. This presentation is misleading to the degree that it gives the impression of a simple, linear implementation process moving from analysis to implementation to performance evaluation, resulting in the presented speedup. Rather, we began with implementing a naive code generator that produced code compatible with the projection engine. The code generator was naive in the sense that whenever faced with a design choice, we chose the design easiest to implement. Then, we implemented a significant portion of DSL code to test the correctness of the implementation. Then, we improved the implementation by manually inspecting generated code and by running different performance experiments using Benchmark.NET [74]. This work included designing and implementing an internal intermediate language (IL) to ease implementation, improving the generated data structures, generating a *virtual portfolio state* (see Section 4.3.4), constant identification, and experimenting with evaluation patterns for the generated code. In parallel with this experimental work, the generator itself had to evolve to remain compatible with the projection platform and more significantly, so did the implemented DSL code (Section 3.2.4).

4.3.3 Code Generation Overview

The code generator transforms a MAL program given as an abstract syntax tree (AST) into a C# program in three steps (Figure 4.1). First, the *Analyser* checks that the program is well-formed by both validating the program's data model, typing its statements and expressions, and making an initialisation check (Section 2.3.1). Second, the *IL Generator* transforms statements and expressions into statements and expressions in an intermediate language (IL) close to C#. All optimisations, such as deforestation, in-lining, vectorisation, and extracting loop constants, occurs in the *IL Generator*. Since C# was the target language, we worked on high-level optimisations since .NET's JIT compiler handles low-level optimisations such as register allocation. We parametrised the code generator with the optimisations it should perform to allow for experimentation. Third, the *C# Generator* creates the final C# program. This generation both consists of a) creating interfaces, classes, and *TypeSpans* and b) making a simple transformation from IL to C#. A *TypeSpan* (a MAL concept) is a collection of entities that has constant time type filtering. Although the IL Generator could hypothetically create C# code directly, we found it useful to introduce the intermediate language to separate optimisations from concerns relating to producing valid C#.

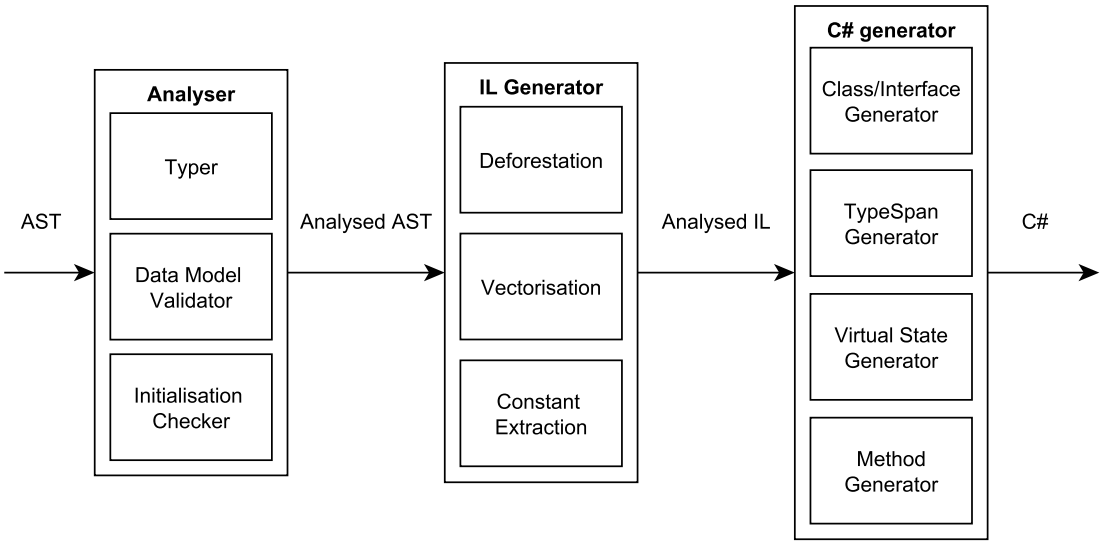

Figure 4.1: A depiction of the code generation process starting (to the left) from a MAL abstract syntax tree (AST) and ending (to the right) with C#. We use the term *Analysed AST* and *Analysed IL* to indicate the program is augmented with analysis information, such as types.

4.3.4 Data Representation Challenges

We encountered the following two engineering challenges in representing MAL's data model in C#. First, we found that it was natural and efficient to represent MAL's data definitions as C# classes. However, Edlund's projection engine internally represents the portfolio state in identifier-indexed dictionaries, as presented in Section 2.2. This means that our first naive code generator produced code having to translate between these two representations whenever computations switched between management actions and the projection engine. To eliminate this overhead, we changed the code generator to create and generate what we call a *virtual portfolio state*. To the projection engine, the virtual portfolio state appears as a collection of identifier-indexed dictionaries. However, internally, they index into MAL's generated classes.

Second, MAL has union types that allow the user to state that a value is one of several types. For example, the type {`OneStatePolicy`, `ThreeStatePolicy`} should be read as the type that contains the two values: `OneStatePolicy` and `ThreeStatePolicy`. When working with a value of a union type, the user is allowed to manipulate fields shared by the possible values. We found that the simplest way of representing union types in C# was through interfaces. However, we found that our first naive implementation that generated interfaces for all possible union types suffered from being inefficient when combined with type downcasts (to a

Table 4.1: Benchmarks from executing MAL in a realistic production cloud environment. The benchmarks were performed on *Standard_F2s_v2* machines with either an Intel Xeon Platinum 8370C, an Intel Xeon Platinum 8272CL, or an Intel Xeon Platinum 8168 processor and 4 GiB of memory.

		1k policies 1k ES	**10k policies 1 ES**
C#	Management	3,560 s	28.9 s
MAL	Management	2,435 s	22.8 s
Speedup	Management	1.46×	1.27×
C#	Full Projection	15,943 s	120.3 s
MAL	Full Projection	14,486 s	109.8 s
Speedup	Full Projection	1.1×	1.09×

more precise type). Generating interfaces only for union types actually used in a program led to a significant performance improvement of generated code.

4.3.5 Benefits of Code Generation

We see three benefits of introducing code generation to the projection platform, namely: runtime performance, interface flexibility, and control over external code. We treat each topic separately.

Runtime Performance
We have benchmarked MAL's generated code against comparable handwritten C# code (Section 2.2) using a realistic production setup. These benchmarks show that MAL's generated code is between 1.27× and 1.46× faster than comparable handwritten C# (Table 4.1). From our experiments, we ascribe the speedup to the following two factors: First, MAL comparatively has a lower memory allocation rate (MB/sec) since the code generator can deforest many computations and avoid creating intermediate collections. Second, MAL generates code that performs constant time type-filtering on collections which is an often-used operation in management actions.

Interface Flexibility
The evolution of the projection engine's interfaces forced the implementation of MAL to evolve too. However, this evolution of MAL often did not require changes

to MAL programs when only the code generator had to be adjusted. For the maintainer of the projection engine, MAL can allow for some changes to interface specifications by providing a layer of separation that can hide these changes from the end-user.

More Control of Performance of Customers' Solutions
Edlund has spent considerable effort on optimising the performance of the projection platform. This optimisation has primarily occurred in the projection engine since this is the part of the platform Edlund is in full control over. On the contrary, it is difficult for Edlund to optimise management actions since they either a) are developed by a customer or b) are defined in template code that should remain relatively stable. From this perspective, MAL's code generator would allow Edlund to further experiment with generating more optimised code. For instance, there are several unexplored ideas, such as loop merging, that could increase the performance of compiled code. Also, from a security perspective, this control is relevant in that it prevents customers from executing arbitrary C#.

4.4 Management Action Language as a Product

We have so far argued that MAL's implementation is mature enough to transition to an Edlund product (Section 4.2), and we have argued for benefits of using MAL in terms of being more declarative (Section 2.2), providing safety guarantees (Section 2.3), being more usable (Chapter 3), and generating efficient code (Section 4.3.5). Still, it currently seems unlikely that MAL will transition from a research project to an Edlund product. We previously set out to analyse why this transition is difficult in an extended journal version of *On Designing Applied DSLs for Non-Programming Experts in Evolving Domains* [48](Appendix A). This analysis is to undergo a *major revision* based on reviewer feedback. Still, we rewrite parts of the extension here because we find it important to disseminate cases where a DSL encounters problems in being adopted.

First, we describe how MAL's business context, and thereby MAL's purpose, has developed through the project. Second, we present a cost/benefit analysis of using MAL. This analysis expresses the problems Edlund and we perceive in transitioning to MAL rather than an objective measure of costs and benefits. Finally, we discuss MAL's transition to a product from the perspective of Tomassetti's and Zaytsev's reflection paper [55] on the lack of DSL adoption and from the perspective of academia-industry collaboration.

4.4.1 Development in Business Context

Edlund's projection platform was created at a time when all Danish pension companies were investigating how they would perform and document balance projections in accordance with the Danish Financial Supervisory Authority's (FSA's) administrative instructions. The potential customers of the new platform were either existing customers of other products, users of competitors' software, or customers who made their own in-house development. Therefore, the platform was designed so that existing customers could use it with other existing products and potential customers could use it with other third-party or in-house software products.

During the beginning of the development of the projection platform, it was in serious competition to obtain an initial market share. The initial market share was vital since it takes a substantial effort to persuade a company to change to a new product after their initial choice. Edlund sought to demonstrate that its new platform was sound both in an actuarial mathematical and a technological sense to obtain an initial market share. Therefore, an early minimal viable product was developed, which could be refined to customer-specific needs using an agile development process. Part of this product was a management template for a realistic pension company, demonstrating some of the product's potential.

As the management template co-evolved with the projection platform, the potential for a DSL became apparent. The management template expanded to demonstrate new functionality and accommodate specific customer wishes. While the projection platform was being developed, pension companies investigated how to use it for their purpose. This investigation involved analysing the requirements from the Danish FSA, developing mathematical models of the company, understanding the projection platform, and acquiring employees capable of using the platform through hiring and training. From this perspective, a DSL could not only be a technical solution but a selling point since it could make the projection platform more attractive and accessible to customers. The DSL could make it easier for customers to acquire employees capable of using the platform and make the platform accessible to more diverse pension companies by making them easier to model.

As we designed and developed MAL, the projection platform's market settled and matured. Pension companies decided on how they would meet the requirements from the Danish FSA, and in doing so, the companies both settled on a technological platform and started their initial exploration of specifying management actions. This development meant that now when MAL has reached a maturity where it has merits, its business potential has moved more from attracting a diverse set of

potential customers towards improving the situation for existing ones.

4.4.2 Costs and Benefits

We analyse the costs and benefits of transitioning MAL to a product in this new business context. We do so from the perspective of both Edlund and Edlund's customers. This analysis has been discussed with part of Edlund's management, and while the analysis does not seek to quantify costs and benefits monetarily, it reflects our shared understanding in terms of *implementation*, *performance*, and *reputation* costs. These costs are summarised in Table 4.2.

Edlund's perspective

Initially, Edlund will have a substantial *implementation cost* of turning MAL into a part of the projection platform and thereby a product. This cost includes a) improving MAL as a product in terms of integration, improved language documentation, improved language environment, and general quality assurance of MAL, and b) current templates written in MAL must be expanded to include all functionality of their existing general-purpose counterparts. However, MAL also alleviates some implementation costs from the outset since MAL's code generator introduces the possibility of evolving company generic parts of the projection engine without the customer noticing. Also, MAL makes it easier to experiment with code optimisations and other forms of code generation and could be a selling point for future customers. There is a risk of a *reputation cost* since some customers would also have to be convinced to switch from their GPL language to MAL. This could impact Edlund's customer relationship even if there are benefits from the onset to customers.

Recurringly, Edlund will have a long-term *implementation cost* by committing to maintain the MAL implementation (parser, type checker, code generator) and documentation. Part of this obligation is to acquire or keep employees with competencies to do so. Another part of this obligation is to evolve MAL as needed and maintain template code akin to a standard library in MAL for customers to use. However, from an implementation perspective, there are also several recurring benefits of using MAL. First, MAL provides more freedom in restructuring, evolving, and optimising the platform, especially if they manage to convince all customers to use MAL. Second, MAL allows for greater modularisation of template code that is easier to maintain. Third, MAL could alleviate friction when onboarding new customers if the customer's wishes are to be implemented in the template solution.

Customer's Perspective

Customers can roughly be separated into three categories represented by customer type A, customer type B, and customer type C. A *type A* customer has strong internal programming resources that develop its management action from scratch. A *type B* customer modifies the template solution to fit its needs. A *type C* customer uses the template solution as-is and requests modifications of the template solution whenever needed.

Initially, there is no *implementation cost* for type C customers, who could simply use the MAL templates as-is. For type B customers, there is some implementation since they must make the same modification to the MAL template as they have made to the existing template. For type A customers, there is a high implementation cost since they must rewrite their existing solution from scratch. In addition, both type A and B customers must acquire some MAL competencies. However, both can expect that it is easier to work in MAL templates once they acquire said competencies. Both type B and type C customers can expect a *performance benefit* since they can expect an immediate 1.27×–1.46× speedup in management action code and corresponding savings in computation time and costs. Type C customers can expect a similar speedup if the performance of their existing management action code is comparable to the template solution.

Recurringly, customers can expect what could be considered the traditional benefits of working in a DSL with a higher level of abstraction. That is lower learning costs, higher programmer efficiency, and more secure, efficient programs. In addition, template-based customers of type B or type C will benefit from using MAL's module system that makes module contracts explicit. More speculatively, using MAL increases the possibility of using code for internal communication or communication with the Danish FSA. There are also downsides to using MAL. First, while MAL's limited expressiveness is one of its strengths, it is also a potential problem. It is likely that some customers will need MAL to be more expressive, and when this happens, they need to convince Edlund to expand the language. Although it is possible to invoke general-purpose procedures from MAL, the point of using MAL disappears if such invocations are regularly needed. A recent example of such new functionality is creating unbounded loops that terminate upon some convergence measure. Second, a customer could fear that using MAL makes it more challenging to migrate to another system in the future. However, such a coupling already exists with the current GPL solutions, and the coupling could even be lower if MAL were to be launched as an open-source project where other vendors could contribute with code generators.

Table 4.2: Summary of expected cost and benefits of using MAL. All costs and benefits are ranked from low to high. Empty cell occurs where the cost is not relevant. The reputation cost is not relevant for any customer type.

	Initial		**Recurring**	
	Benefit	**Cost**	**Benefit**	**Cost**
Edlund				
- *Implementation*	Low	High	Medium	Low
- *Performance*	Low		Medium	
- *Reputation*		Medium		
Customer type A				
- *Implementation*	Low	High	Low	Low
- *Performance*	Low		Low	
Customer type B				
- *Implementation*	Medium	Medium	Medium	
- *Performance*	Low		Low	
Customer type C				
- *Implementation*				
- *Performance*	Low		Low	

4.4.3 Challenges to Adopting DSLs

In their aforementioned paper, Tomassetti and Zaytsev identify and discuss reasons for the lack of more mainstream adoption of DSLs based on anecdotal experiences and community discussions. Although MAL is in a situation where the language has already been developed, we find that the paper describes several challenges we have experienced and discuss them here. In particular, it is perceived to be *risky* to introduce MAL to customers as a product. This is possibly a valid perception, with a deadline for implementing the new financial regulations approaching in 2023. The following discussed factors contribute to the perception that it is risky to introduce MAL.

First, the general lack of DSL adoption makes management more cautious in choosing to adopt a DSL. That is, using MAL is not the neutral option for expressing management actions, whereas using a general-purpose language is (Section 4.3.1). Therefore, we have had to argue for the benefits of using MAL, although it is easy to point to problems with the current neutral solution of using C#.

Second, adopting MAL would require Edlund to have competencies to maintain and evolve the language. This is a long-term obligation that requires retaining current employees and acquiring future employees both with such competencies. This means that adopting MAL is not a question of only software engineering but also human resources.

Third, creating MAL in a collaborative research project introduces some risks for Edlund, even though MAL was created under a cooperation agreement explicitly addressing potential licensing issues. We hypothesise that some of the cost of transitioning MAL to a product could have been mitigated if the language had been introduced to customers at an earlier stage. However, from the perspective of Edlund, this advice was not straightforward to follow due to the nature of collaborating with academia. When starting developing the projection platform, Edlund saw the potential utility of creating a DSL as a way to handle the software variability of their new product. While the company pursued this idea within the setting of academia and industry collaboration, it seemed necessary to have an earlier alternative solution for three reasons. First, at the start of the project, Edlund did not know the PhD fellow who would work on the project or how the project would turn out. Second, Edlund wanted to demonstrate an early working projection platform for potential customers and needed management action specifications for this purpose. Third, Edlund needed an early way to experiment with the projection platform to investigate the mathematics of balance projections. A solution to this problem would be the primary instigator of creating the DSL, but that would question the purpose of the collaboration.

4.5 Summary

In this chapter, we have described our work with implementing MAL and, specifically, its code generator. First, we have described why it is important for MAL to generate efficient code due to the computational cost of large-scale balance projections. Second, we describe how our experimental work with implementing MAL's code generator. We have described the overall process of code generation and how generated code is around 1.3× faster than comparable handwritten C# in a production setup. In [73](Appendix C), we describe the generation in greater detail with translation schemes for generating TypeSpans and interfaces for union types. Finally, we have discussed the how MAL's business context has developed and our experienced challenges in seeking to transition MAL to a product through a cost-benefit analysis.

Chapter 5

Life Cycles of DSLs

DSLs have been used within many different domains to solve a wide variety of problems. This work has led to papers demonstrating the domain utility of a newly created DSL and its contribution to the DSL field. However, these papers leave the reader wondering what happened to the DSL after its initial creation and publication. In this chapter, we set out to mitigate this problem. We discuss the findings and limitations of our paper *Survey of Established Practices in the Life Cycle of Domain-Specific Languages* [64](Appendix D), which examine the life cycle of 30 DSLs through a questionnaire.

5.1 Survey Study on DSLs' Life Cycle

The diverse set of DSLs and publications on DSLs has led other scholars to survey the field to establish the current state of the field. These surveys either look to the DSLs themselves or the related publications as the primary source of information. Both investigation methods come with strengths and weaknesses. Analysing DSLs themselves gives unbiased information on the DSLs and their features but gives no information on the development process and usage. Analysing publications may give information on the development process and usage but only to the degree that authors report the subject of interest. Also, publication analysis relies on the self-reporting by authors. To mention some examples of both kinds of analysis and the subject the studies explore: Dragule et al. survey the design space of DSLs for robotic missions by examining 30 programming environments for robotic missions [75]. Kapre and Bayliss find key features of DSLs used for high-performance field-programmable gate array computing through the survey of 9 DSLs [76]. Nascimento et al. [5] and Kosar et al. [6] conduct systematic mapping studies to investigate research within the field. Iung et al. [38] conducted a similar systematic mapping study for tools being used by DSL creators.

5.2 Purpose and Method

The original idea of our survey was to investigate what has happened to influential DSLs after their publication. Specifically, we wondered what has happened to DSLs appearing in the classical paper *Domain-Specific Languages: An Annotated Bibliography* [77]. From this idea, we set out to survey established practices in the life cycle of DSLs by e-mailing (or by other means of internet contacting) a questionnaire to DSLs appearing in various curated DSL collections. The questionnaire focused on the user perspective, design process, and evolution, since these subjects are a) difficult to examine by analysing publications and b) major themes in this thesis. To our knowledge, our survey using a questionnaire is the first of its kind.

Our chosen method of sending a questionnaire to selected authors has its own methodological weaknesses. First, there is a selection bias in that we were not able to contact all authors, and not all authors answered the questionnaire. From this perspective, we expect a survivorship bias since authors of successful DSLs are both easier to contact and more likely to answer. Also, we expect that authors of newer publications are easier to contact (since contact information appearing in a publication is more likely up-to-date). Second, we rely solely on recipients' interpretation of our questions and their self-reported answers. Even though we found it unlikely that recipients maliciously answered the questionnaire, it may be difficult for authors to remember the exact details of a project they began a decade or more ago. Our primary mitigation to these issues was to a) try different ways of contacting authors, b) make it easy to answer the questionnaire and skip questions, and c) discuss the wording of questions with colleagues beforehand.

We obtained a response rate of 43% from authors who could have received the questionnaire, that is, from invitations sent to authors that did not get an error response from a mail server. We are happy with the response rate since we find that a high response rate minimises the risk of a selection bias in who responded. We ascribe some of the response rate to every author receiving a personalised questionnaire invitation that specifically mentions both the author's DSL(s) and the author's publication(s). We also asked whether we could contact the authors for clarification or small interviews as a form of further investigation.

5.3 Findings in Context of other Publications

We summarise and discuss our findings that directly relate to other aspects of our work. Our DSL sample had the following characteristics. The average age of the 30 DSLs was 11 years (introduced in 2010), with 80% being younger than 12 years.

Allowing multiple answers, 77% of the DSLs were developed within an academic setting, 38% were developed in an industrial setting, and 30% were developed in an open-source community. The most common purpose for creating the DSL was "to improve program conciseness and readability", selected by 50% of respondents. However, only 3% had this as their only option.

5.3.1 No Established Practice for User Involvement

We attempted to elicit practices for involving users in the development process. Such practices are relevant to our own work with human-centred design and co-design. However, we find that such practices are difficult to examine through analysis of publications since far from all authors report on user involvement. Whenever authors do not make such reporting, one cannot know whether it is because there was no user involvement or whether the authors did not find user involvement to be an important part of their work. Through the survey, we found:

1. DSLs are developed for users with all levels of programming experience (from none to expert).
2. DSLs are designed with all levels of user involvement (from none to all decisions).
3. We found no correlation between the level of user involvement and programming experience (or year of DSL introduction).
4. Prospective users were most commonly involved in requirement elicitation and feedback.
5. In some projects, users are part of the design team; in other projects, they are not involved at all.
6. When asked broadly about their thoughts on their entire project, authors did not report dissatisfaction with how users had been involved.

In summary, there are widely different approaches to involving users in a DSL project. Still, most practitioners are not dissatisfied with their approach. These findings leave one wondering how there can be such diverse approaches with little dissatisfaction. We hypothesise that the following two circumstances both contribute:

(a) The level of user involvement must be seen within the specific setting of a design project. In some projects, the amount of user involvement may be dictated by the settings, so there was no choice for the practitioner and, thus, little dissatisfaction. For example, users may be either unavailable or, at the opposite end of the spectrum, part of the design team.

(b) Respondents answer within the frame of their own choices regarding the design process. Proponents of user involvement are more likely to have and like user involvement, and conversely for sceptics of user involvement. Also, proponents of user involvement may overestimate the utility of user involvement, and sceptics may underestimate the utility of user involvement.

5.3.2 Evolution is Pervasive and Important

In our paper [48](Appendix A and Section 3.2.2), we presented our categorisation of programming languages based on their evolutionary characteristics. We hypothesised that many DSLs belong to the class of rapidly evolving E-type languages due to the DSL affecting the application domains they are designed for. For example, launching a DSL likely causes a response from users having ideas about new aspects of the application domain or features not included in the original design. Such user feedback may lead to the evolution of the DSL, which again causes a new user response.

Although the survey was not meant to empirically test the hypothesis, we find the following empirical evidence for our characterisation of these DSLs:

- The vast majority of surveyed DSLs had experienced evolution.
- The evolution of a DSL affects its success.
- Allowing multiple answers, 47% of the DSLs evolved due to new user wishes, 40% evolved due to new areas of application, and 13% evolved due to changes in the existing application domain.

Still, we also found some evidence that questions aspects of the characterisation of DSLs' (and E-languages') causes for evolution, since 40% of DSLs evolve due to changes in the implementation and 37% evolve due to changes in external technology. It is unlikely that the DSLs' introduction to an application domain affected these two causes of evolution.

5.3.3 Pragmatic Development

We found pragmatism to be a reappearing theme in several of the survey findings. While pragmatism is not an explicit theme in our work so far, we see it as a reappearing underlying theme and address it in Section 7.2. From this perspective, we find it relevant to display how the theme appears in the survey. Here, we should

clarify that we use the term *pragmatic* as the opposite of *idealistic*.

We find that respondents report using different kinds of flexibility allowed by their project in a pragmatic manner. First, several respondents reported that they have had to make breaking changes to their language, especially early in its life. Second, some respondents reported that finding new application domains for their DSL was important for its success. Third, respondents reported major improvements made to the DSL and its launch in terms of performance, usability, and modularity.

5.4 Summary

In this chapter, we have presented our survey that set out to investigate practices in the life cycle of DSLs. We used a questionnaire since it provided the best method of investigation as it allowed for fine-grained investigation on subjects of interest when compared to surveying articles or DSLs themselves. We have discussed our findings regarding how users are involved in a project, the evolution of DSL, and pragmatism in approach in relation to work presented in previous chapters.

Chapter 6

Secrecy Analysis in Distributed Workflows

With Mads Frederik Madsen and Søren Debois

We take a detour to present ongoing work within the business process modelling language called Dynamic Condition Response (DCR) graphs [78], where we consider how to implement secrecy guarantees to the language. DCR graphs allow for a high-level business process description that can facilitate collaboration between adversarial actors [79]. Due to the difference in goals, an actor in a multi-actor business process may want aspects of their activities to remain secret to others. Mads Frederik Madsen and Søren Debois have ongoing work defining and investigating secrecy and indistinguishability in DCR graphs from a primarily theoretical perspective[1]. We reiterate some of this work as background information (while not claiming any ownership) since it is needed for the chapter. In this chapter, we consider a more practice-oriented extension of their previous work by investigating the possibility of statically providing secrecy guarantees (the latter part of Section 6.3 and onwards) for DCR graphs. Here, we give an informal presentation of our approach that is formalised in our draft paper *Static Secrecy Guarantees for Dynamic Condition Response Graphs* [80](Appendix E).

6.1 Distributed Dynamic Condition Response Graphs

A distributed DCR graph consists of a set of *events*, a set of *actors*, a set of *relations*, and a *marking* describing the state. We will use the example DCR graph depicted in Figure 6.1 for a crash course in DCR graphs and as a running example. The *events* (`Event A`, `Event B`, `Event C`, and `Event D`) are depicted as nodes. Each event

[1]Since this is ongoing, work we do not have a published reference at the time of writing

belongs to one of the three *actors* (`Alice`, `Bob`, and `Carol`), which is annotated at the top of each event node. In the secrecy analysis, we consider four types of *relations* (`include`, `exclude`, `condition`, and `response`). Here, we only present the `include` and `exclude` relation. In the example, there are four `include` relations denoted by a green arrow with a + head (e.g., `Event A` includes `Event B`) and one exclude relation denoted by a red arrow with a % head (e.g., `Event D` excludes `Event A`). Each event has a marking that describes its state that is known only to its actor. The *marking* of an event describes whether it

- has been executed (denoted by a checkmark, e.g., `Event D`, but not `Event A` in Figure 6.1)
- is enabled or disabled (denoted by a white or grey background, respectively), meaning whether the event can be executed or not.
- is included or excluded (denoted by a solid or dashed border, respectively), which here corresponds exactly to the enabled/disabled state since we only consider `include` and `exclude` relations.

In the example, `Event A` can be executed since it is enabled. The result executing `Event A` is depicted in Figure 6.2. The changes from Figure 6.1 to Figure 6.2 is that:

- `Event A` has become executed.
- `Event B` and `Event C` have become included (and thereby enabled) due to the two outgoing `include` relations from `Event A`.
- No events have become excluded since `Event A` does not have any outgoing exclude relations.

A DCR graph is a high-level process specification that has a translation into a low-level specification as a labelled transition system (LTS). The LTS representation describes (and defines) a DCR graph's dynamic behaviour as a graph that has *markings as nodes* and *event executions as transitions* (or directed edges). Figure 6.3 shows a fragment of the LTS corresponding to the DCR graph depicted in Figure 6.1. The following procedure translates a DCR graph to an LTS:

1. Create an LTS node for each reachable marking of the graph.
2. Create a transition between a source and target node if it is possible to execute an event in marking represented by the source node, and doing so leads to marking represented by the target node.

The cost of this translation is at worst exponential in the number of events since a DCR graph's number of possible markings is exponentially bounded by its number of events.

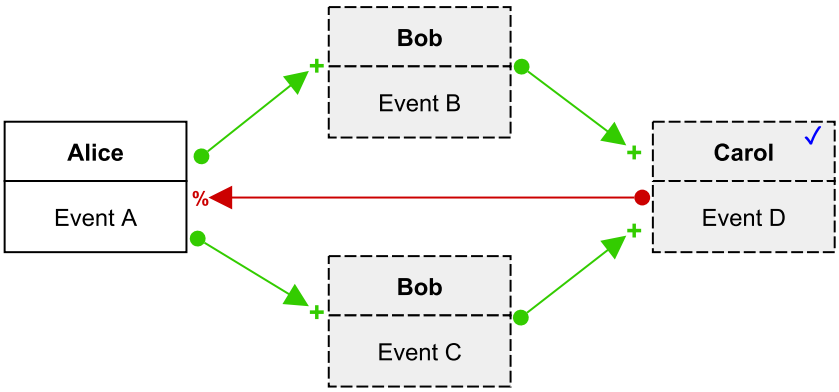

Figure 6.1: The DCR graph that we will use as our motivating example in this chapter in the standard visual notation. The graph consists of four events, each belonging to one of three actors.

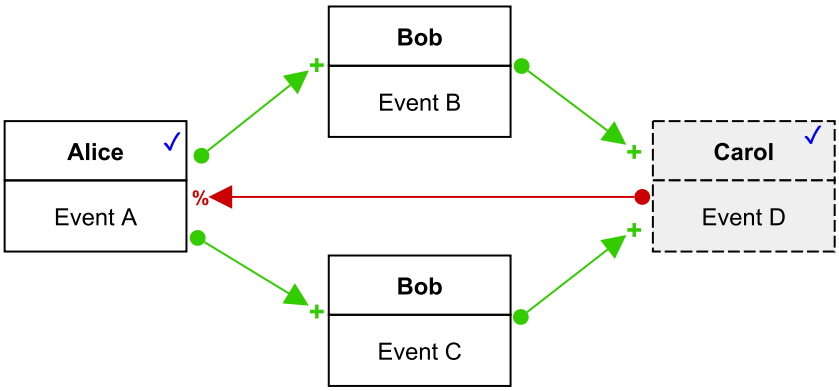

Figure 6.2: The DCR graph that is obtained by executing `Event A` in Figure 6.1. `Event A` has become executed. `Event B` and `Event C` have become included and enabled.

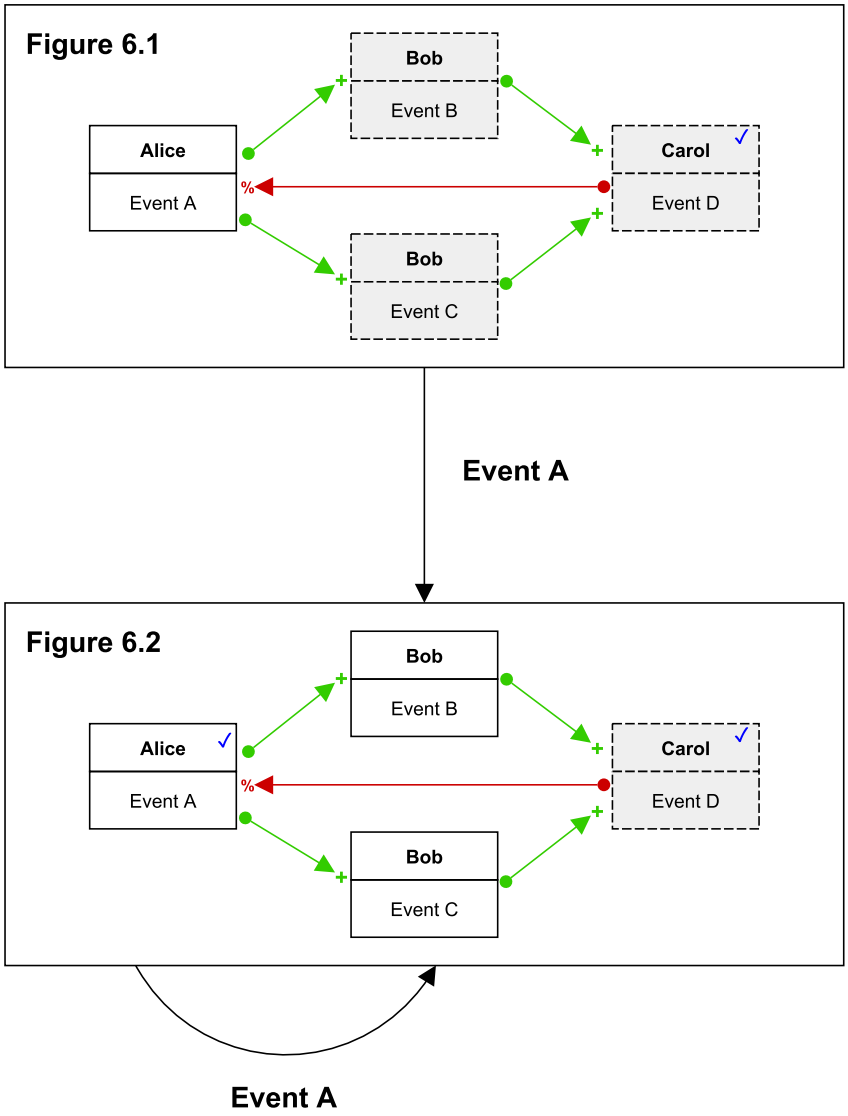

Figure 6.3: A fragment of the labelled transition system (LTS) obtained from the DCR graph depicted in Figure 6.1. The LTS fragment contains two nodes representing the markings of Figure 6.1 and Figure 6.2 and two edges representing the changes in marking from executing `Event A`. The full LTS has additional outgoing transitions from the Figure 6.2 node corresponding to the execution of `Event B` and `Event C`.

6.2 DCR Knowledge

Using Figure 6.1, we can give the following informal examples of *knowledge* and *secrecy*. If `Bob` at some point observes `Event B` changing its state to included then he knows that `Alice` has executed `Event A`. He may make this inference since there is only one incoming include relation to `Event B`, and it originates from `Event A`. If `Carol` at some point observes `Event D` changing its state to included she knows that `Bob` has executed either `Event B` or `Event C`. She may make this inference since there are two incoming include relations to `Event D` originating from respectively `Event B` and `Event C`. However, it remains secret to `Carol` whether `Bob` has executed `Event B` since she can never know for certain that `Bob` has executed `Event B`. It is known to `Carol` that `Alice` has executed `Event A` by the following line of reasoning. `Alice` must execute `Event A` for `Bob` to execute `Event B` or `Event C`. It is known to `Carol` that `Bob` has executed either `Event B` or `Event C`, and therefore, it is also known to her that `Alice` has executed `Event A`.

A bit less informally, we are interested in reasoning about whether an actor knows that a property $P : \texttt{marking} \rightarrow \texttt{bool}$ is true for a marking at some point during execution. We say that actor A knows P iff there exists a sequence of event executions that leads to a marking where P is inferable to be true by A. Since A only observes the part of the marking that describes her event, A may not know that P is true in a marking even if it is. Conversely, we say that P is secret to A iff A does not know P, i.e., there does not exist a sequence of event executions that make P inferable by A. Even more formally, we use a notion of knowledge similar to that of epistemic logic as presented in [81].

6.3 Static Analysis

As the above examples demonstrate, it quickly becomes complicated, error-prone, and cumbersome to reason about actors' secrecy and knowledge, even in a simple example. Therefore, we seek a static analysis of secrecy that is both efficient and correct. If we do not care about efficiency, we analyse secrecy for a given DCR graph and actor through the following procedure:

1. Translate the DCR graph to its LTS representation.

2. Group the markings of executions that the actor cannot distinguish[2] from

[2]Since we are rather informal in this chapter, we use the informal term *cannot distinguish*. To give a more precise intuition, observe how Carol cannot observe a state change when transitioning between the two nodes of Figure 6.3. To her, these two markings are indistinguishable.

each other (the markings of a group are indistinguishable to the actor).

3. If there exists a group where some property P holds for all markings, then P is known to the actor.

While step 2 is not completely trivial due to the possibility of infinite runs, the above procedure lets us reason precisely about knowledge and secrecy. However, the exponential translation to an LTS representation makes this approach computationally infeasible for large DCR graphs.

Instead, inspired by information flow analysis [82], we seek a sound approximation of secrecy that is computationally feasible. A crude approximation of secrecy can be obtained by letting knowledge flow over all relations of a DCR graph by the following procedure:

1. Initially, consider all events to know everything about their own state (and implicitly the effect of executing the event).

2. Iteratively, let knowledge flow over `include` and `exclude` relations such that all knowledge flow from the source event of a relation to its target event.

3. Terminate when an iteration does not cause any change to the knowledge of events.

Table 6.1 shows all steps of the crude approximation for the DCR graph in Figure 6.1. Since there is a finite amount of knowledge in the system, the above procedure terminates. While this approach is computationally feasible, it is unsatisfying in that it is too crude in overestimating knowledge. For example, the analysis erroneously would state that `Carol` knows whether `Bob` has executed `Event B`, although this property is secret, as argued in Section 6.2.

6.4 Improvements and Challenges

While the afore-mentioned crude knowledge approximation is not completely trivial, we find it too crude to be useful. It is essentially a glorified graph reachability analysis stating that if `Event B` is reachable by `Event A`, then the owner of `Event B` knows everything about `Event A`. Therefore, we seek to strike a balance between completeness and efficiency that is more complete while remaining sound.

Our current goal of completeness is to be able to reason about uncertainty by using logic disjunction ("or"), e.g., Carol knows `Event B` *or* `Event C` was executed. Uncertainty can be introduced to the flow analysis by making a disjunction for

Table 6.1: A table showing the knowledge of each event for different iterations of our crude knowledge approximation of the DCR graph in Figure 6.1. We use `E` to mean knowledge of event `E` being executed and `*` knowledge of all events.

Step	Event A	Event B	Event C	Event D
0	A	B	C	D
1	A,D	A,B	A,C	B,C,D
2	*	A,B,D	A,C,D	*
3	*	*	*	*

knowledge incoming from identical relations. However, doing so introduces two problems to the approximation. First, the approximation may now not terminate due to disjunctions growing infinitely. We solve this by having a flat set representation of logic disjunction where elements of the set represent worlds an actor cannot distinguish. Second, when having disjunctions, there is a need to reason about events being prevented from executing since an actor may use such information for inference. For example, Carol knows `Event B` *or* `Event C` was executed. Carol also knows that `Event B` has been prevented from executing. Therefore, she may conclude that `Event C` was executed.

In our paper [80](Appendix E), we present our secrecy approximation in its current form and our work *towards* proving that our approximation is sound, meaning it may report false negatives but not false positives on secrecy. While we believe it to be sound, previous proof efforts have uncovered problems with previous versions of the secrecy approximation. In working with the proof, many details show up, such as a) handling `response` and `condition` relations, b) making inferences from knowing that an event is disabled from executing, and c) considering knowledge obtained from different orderings of event execution. These details mean that a substantial amount of work has been put into defining *secrecy approximations* and the concept of *minimal runs*[3] and proving that it is a secrecy approximation, which we can continue to refine.

[3]A run is a sequence of event executions

6.5 Summary

In this chapter, we have argued for the necessity of approximating secrecy in DCR graphs since an exact analysis is computationally infeasible. We have given an informal presentation of a crude version of our approximation that lets information flow over DCR relations. Still, we seek a better approximation that allows us to reason about uncertainty in actor knowledge. However, introducing uncertainty comes with new problems since it requires us to reason about events being *prevented* from being executed instead of just events being executed. Due to the complexity of our approximation, we see this work as incomplete until we have a formal proof showing that our approximation of secrecy is sound.

Chapter 7

Conclusion

In this thesis, we have presented our different work emerging from a project that set out to design and implement a DSL for the Danish pension industry. Besides demonstrating the feasibility and utility of such a domain-specific language (DSL), this work falls into three general areas. First, we have explored the possibility of using human-centred design and co-design to create DSLs in the context of the Management Action Language (MAL). Here we found that when we adapted these design methods to our design context, they led to valuable findings. Second, we have explored the implementation of DSLs both in the context of MAL and in the context of Dynamic Condition Response (DCR) graphs. For MAL, an important aspect of the implementation was to generate efficient code due to the computational cost of balance projections. For DCR graphs, we presented ongoing work towards statically and soundly providing secrecy guarantees. Third, we have investigated the broader life cycle of DSLs beyond design and implementation. We sent a questionnaire to creators of DSLs appearing in different curated DSL collections to establish the current state of practices in topics such as user involvement and evolution. Also, we analysed and discussed the problems MAL is facing in transitioning from a research project to a product.

We find that a theme of evolution and pragmatism reappear in many of the discussed topics. It is not by accident that the themes have reappeared since we have found them interesting and chosen to investigate them. Still, we find it worth concluding on these two themes on their own.

7.1 Evolution

In hindsight, it was almost inevitable that MAL's domain and MAL itself were going to evolve during this PhD project since the language was being designed for a new projection platform being incrementally developed for a new domain.

However, this was not at the forefront of our mind when setting out to design MAL; instead, usability and user evaluation were.

In our paper [48](Appendix A), the evolution we experienced led us to create a classification of programming languages according to their evolutionary characteristics (Section 3.2.2). When revisiting this classification, we consider whether it would be more meaningful to classify *causes* of programming language evolution instead of the *programming languages* themselves. This consideration originates from experiencing that many programming languages are somewhat *P-type* and somewhat *E-type*. Instead of getting hung up on this difference, we find it more useful to a) discuss underlying causes of evolution in terms of *P-type* or *E-type* languages and b) discuss what kind of mitigations there are for different causes of evolution. From this perspective, the classification is similar in spirit to the DSL typology proposed in Section 3.3.

In our paper [64](Appendix D), we surveyed other DSL creators' experiences in the life cycle of their DSL. We have already discussed some of these findings in relation to the proposed classification in Section 5.3. While we found that some of these findings supported classifying application-domain oriented DSLs as *E-type* languages, we also found that it is common for DSLs to evolve due to changes in their implementation technology. This cause for evolution is neither captured by *S-type*, *P-type*, or *E-type* language categories.

In our paper [64](Appendix E), we presented our ongoing work toward providing secrecy guarantees for Dynamic Condition Response (DCR) graphs. We see this work as a clear example of *E-type* evolution as follows: DCR graphs provide a declarative and high-level notation for modelling multi-actor business processes. This notation led to work considering and demonstrating the viability of using DCR graphs to let adversarial actors cooperate in well-defined business processes. Again, this expected use of the notation led to considerations of the precise meaning of secrecy in DCR graphs, which in turn led to our work with approximating secrecy. As such, modelling with DCR graphs caused an evolution of the language implementation because of the utility the notation provided to its application domain.

7.2 Pragmatism

We framed this thesis by prefacing it with a quote by the renowned language designer Anders Hejlsberg that materialises an underlying thesis theme of pragmatism in programming language design. At the IT University of Copenhagen, Hejlsberg

presented work on designing TypeScript [83, 84], which by 2022 has rapidly become one of the most commonly used programming languages worldwide [85, 86, 87]. From an academic perspective, one could prescribe that when someone sets out to design a typed alternative to JavaScript, then they should ensure that the type system is meaningful by proving that it is sound. Instead, Hejlsberg presented the pragmatic goal for TypeScript that it should accept as many existing meaningful JavaScript programs as possible. Hejlsberg was willing to compromise the type-theoretic soundness of TypeScript's type system to reach this goal. In this project, we have experienced similar conflicting goals between the ideal and actual world, leading to pragmatic choices.

In our paper [48](Appendix A), we set out to use user-centred design to create MAL, a language specifically for Danish actuaries. In an ideal world, we would, in each design iteration, evaluate a fully functioning implementation with actual users. However, due to the limited availability of users and the cost of implementing designs, we started out by evaluating text-editor prototypes with in-team experts. Also, in an ideal world, domain experts would be able to define and delimit a domain. However, we found that since the domain was also new to domain experts, then we had to rely on qualified guesses on how the domain would look after further exploration.

In our paper [49](Appendix B), we set out to co-design quality assurance measures with MAL's prospective users. The co-design activity is in itself a pragmatic choice since we ourselves knew what we considered as good practices for quality assurance and knew how we could support these practices. However, we were unsure if our prospective users would agree with our perspective on quality assurance and if they would find our quality assurance measures valuable.

In our paper [64](Appendix D), we investigated the experiences of other DSL creators. As already discussed in Section 5.3, we found that other creators often reported making different pragmatic choices. Most surprisingly, we found a couple of reports on changing the domain of a DSL which almost per definition seems to contradict the original intent of the language. These experiences resulted in us giving the recommendation that DSL creators should use the flexibility allowed by their project, i.e., be pragmatic in their approach to development and design.

Reflecting on this recommendation, it is possible that we should have made some other choices during the creation of MAL. First, we spent a significant effort on proving a subset of MAL to be type safe (Section 2.3). While this effort allowed us to broaden our academic horizons and develop new skills, it had only little

effect on improving MAL's typechecker. Second, we find that it would have been advisable to try to use MAL as a product earlier in the project. Doing so would have meant exposing Edlund's customers to a less complete language design and implementation, which can be viewed as a pragmatic approach to launching a DSL. However, this would have made Edlund reliant on an external developer (namely, me) for maintaining business-critical functionality (Section 4.4).

7.3 Future Work

We see five avenues of future work in the subjects explored in this thesis. Three of these call for further empirical investigations into existing DSLs and GPLs. We treat each avenue on its own.

7.3.1 Problem Space DSL Typology

In Section 3.3, we presented our bid for the creation of a problem space DSL typology for the purpose of helping DSL creators in their methodological design choices. However, the proposed typology is based on our experience, our view of state of the art, and our hypothesising. Future work should refine the (or make an alternate) typology based on empirical explorations. The explorations should examine a) different kinds of DSL dimensions and b) the costs and benefits of applying different design techniques. For example, one could compare the findings of usability tests with the findings of a heuristic cognitive walk-through along different DSL dimensions.

7.3.2 Empirical Investigations into S, P, and E Languages

The presented categorisation of *S-type*, *P-type*, and *E-type* languages should be investigated more closely to see how well it describes the evolution of existing languages. We see two easily accessible sources for information on changes that could be categorised. First, one could examine release notes from established languages. Second, one could examine the changes taking place in open source programming language projects.

7.3.3 Interview DSL Creators on Life Cycle of DSLs

In the survey [64] on the life cycle of DSLs, we used a questionnaire to get uniform answers on topics of interest. However, due to the nature of the questionnaire, many subjects were only investigated at a surface level. The questionnaire should

be followed up by in-depth interviews with some of the DSL creators to validate our interpretation of the findings and to further examine the topics.

7.3.4 Bayesian Modelling of Knowledge in DCR Graphs

We used a non-probabilistic (or possibilistic) model of knowledge when examining secrecy guarantees of DCR graphs in that an actor either does or does not know something. A natural extension of this work would be to use a more Bayesian model of knowledge where an actor model has some prior assumption about others and let the actor update the prior when making observations. This is the setup used in quantitative information flow [88] where one could seek to bound the information leak from an actor observing (and possibly interacting with) the system.

7.3.5 Tool Support for DSL Evaluation

Within the field of language engineering, there is a focus on workbenches and tool support. However, there is a lack of tool support for conducting usability evaluations of programming languages. In psychology, tools such as OpenSesame [89] and iMotions [90] allow researchers to easily set up experiments and choose different measures such as reaction time or eye tracking. Similarly, the field of programming language research should have tools assisting language evaluation. For example, a tool with support for the Language Server Protocol [65] could allow a designer to semantically validate whether a task is solved by test participants while giving advanced measure outputs such as eye-tracking interactions with keywords.

Bibliography

[1] B. M. Jensen, M. D. Raffnsøe, and J. She, "Forsikrings- og pensionssektoren i ny kvartalsvis statistik," 2019, (In English: The Insurance Sector and Pension Sector in New Quarterly Annualy Statistic).

[2] H. S. Borum, "Management Action Language - Design and Technical Description of a Domain Specific Language for Non-programming Professionals," Ph.D. dissertation, IT University of Copenhagen, Denmark, 2020.

[3] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*, S. Apel, D. Batory, C. Kästner, and G. Saake, Eds. Berlin, Heidelberg: Springer, 2013, https://doi.org/10.1007/978-3-642-37521-7.

[4] M. Shaw, "Myths and mythconceptions: what does it mean to be a programming language, anyhow?" *Proceedings of the ACM on Programming Languages*, vol. 4, no. HOPL, pp. 234:1–234:44, Apr. 2022. [Online]. Available: https://doi.org/10.1145/3480947

[5] L. Nascimento, D. Viana, P. Neto, D. Martins, V. Garcia, and S. Meira, "A Systematic Mapping Study on Domain-Specific Languages," Nov. 2012.

[6] T. Kosar, S. Bohra, and M. Mernik, "Domain-Specific Languages: A Systematic Mapping Study," *Information and Software Technology*, vol. 71, pp. 77–91, Mar. 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584915001858

[7] D. Christiansen, K. Grue, H. Niss, P. Sestoft, and K. S. Sigtryggsson, "An Actuarial Programming Language for Life Insurance and Pensions," in *Proceedings of 30th International Congress of Actuaries*, 2013.

[8] H. Borum, "MAL," Jul. 2022, original-date: 2022-07-25T19:17:45Z. [Online]. Available: https://github.com/hborum/MAL

[9] Skatteministeriet, "Bekendtgørelse af pensionsafkastbeskatningsloven," Jun. 2020, library Catalog: Retsinformation. [Online]. Available: https://www.retsinformation.dk/eli/lta/2020/185

[10] "The Agda Wiki," accessed: May 2022. [Online]. Available: https://wiki.portal.chalmers.se/agda/pmwiki.php

[11] A. Barišić, V. Amaral, and M. Goulão, "Usability driven DSL development with USE-ME," *Computer Languages, Systems & Structures*, vol. 51, pp. 118–157, Jan. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1477842417300477

[12] M. Voelter, *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. Lexington, KY: CreateSpace Independent Publishing Platform, Jan. 2013.

[13] A. Barišić, V. Amaral, M. Goulao, and A. Aguiar, "Introducing usability concerns early in the DSL development cycle: FlowSL experience report," p. 10, 2014.

[14] D. Norman, *The Design of Everyday Things: Revised and Expanded Edition*, revised edition ed. New York, New York: Basic Books, Nov. 2013.

[15] S. Gasson, "Human-Centered vs. User-Centered Approaches to Information System Design," p. 18.

[16] K. Bodker, F. Kensing, and J. Simonsen, *Participatory It Design: Designing for Business and Workplace Realities*. Cambridge, MA, USA: MIT Press, 2004, https://doi.org/10.1109/TPC.2005.853942.

[17] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon, "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools," *Computer*, vol. 49, no. 7, pp. 44–52, Jul. 2016.

[18] M. Coblenz, J. Aldrich, B. A. Myers, and J. Sunshine, "Interdisciplinary programming language design," in *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2018. Boston, MA, USA: Association for Computing Machinery, Oct. 2018, pp. 133–146.

[19] M. Maguire, "Methods to support human-centred design," *International Journal of Human-Computer Studies*, vol. 55, no. 4, pp. 587–634, Oct. 2001. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1071581901905038

[20] J. Wong and J. I. Hong, "Making mashups with marmite: towards end-user programming for the web," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. San Jose California USA: ACM, Apr. 2007, pp. 1435–1444. [Online]. Available: https://dl.acm.org/doi/10.1145/1240624.1240842

[21] A. Romero-Garces, L. J. Manso, M. A. Gutierez, R. Cintas, and P. Bustos, "Improving the lifecycle of robotics components using Domain-Specific Languages," *arXiv:1301.6022 [cs]*, Jan. 2013, arXiv: 1301.6022. [Online]. Available: http://arxiv.org/abs/1301.6022

[22] D. Gritzner and J. Greenyer, "Synthesizing Executable PLC Code for Robots from Scenario-Based GR(1) Specifications," in *Software Technologies: Applications and Foundations*, M. Seidl and S. Zschaler, Eds. Cham: Springer International Publishing, 2018, vol. 10748, pp. 247–262, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-319-74730-9_23

[23] P. Detzner, T. Kirks, and J. Jost, "A Novel Task Language for Natural Interaction in Human-Robot Systems for Warehouse Logistics," in *2019 14th International Conference on Computer Science Education (ICCSE)*, Aug. 2019, pp. 725–730, iSSN: 2473-9464.

[24] M. Schuts, "Industrial Experiences in Applying Domain Specific Languages for System Evolution," Ph.D. dissertation, 2017.

[25] B. Hoffmann, K. Chalmers, N. Urquhart, and M. Guckert, "Athos - A Model Driven Approach to Describe and Solve Optimisation Problems: An Application to the Vehicle Routing Problem with Time Windows," in *Proceedings of the 4th ACM International Workshop on Real World Domain Specific Languages*, ser. RWDSL '19. Washington D. C., DC, USA: Association for Computing Machinery, Feb. 2019, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3300111.3300114

[26] R. Halvorsrud, I. M. Haugstveit, and A. Pultier, "Evaluation of a modelling language for customer journeys," in *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Sep. 2016, pp. 40–48, iSSN: 1943-6106.

[27] D. Wile, "Lessons learned from real DSL experiments," *Science of Computer Programming*, vol. 51, no. 3, pp. 265–290, Jun. 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642304000310

[28] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel, "Design Guidelines for Domain Specific Languages," *arXiv:1409.2378 [cs]*, Sep. 2014, arXiv: 1409.2378. [Online]. Available: http://arxiv.org/abs/1409.2378

[29] B. A. Myers, J. F. Pane, and A. Ko, "Natural programming languages and environments," *Communications of the ACM*, vol. 47, no. 9, pp. 47–52, Sep. 2004.

[30] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull, "Exploring language support for immutability," in *Proceedings of the 38th International Conference on Software Engineering*. Austin Texas: ACM, May 2016, pp. 736–747. [Online]. Available: https://dl.acm.org/doi/10.1145/2884781.2884798

[31] J. Sánchez-Cuadrado, J. de Lara, and E. Guerra, "Bottom-Up Meta-Modelling: An Interactive Approach," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds. Berlin, Heidelberg: Springer, 2012, pp. 3–19.

[32] M. Kurhmann, G. Kalus, and A. Knapp, "Rapid Prototyping for Domain-specific Languages - From Stakeholder Analyses to Modelling Tools," *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, vol. 8, no. 1, pp. 62–74, 2013, number: 1. [Online]. Available: https://emisa-journal.org/emisa/article/view/102

[33] J. L. C. Izquierdo, J. Cabot, J. J. López-Fernández, J. S. Cuadrado, E. Guerra, and J. de Lara, "Engaging End-Users in the Collaborative Development of Domain-Specific Modelling Languages," in *Cooperative Design, Visualization, and Engineering*, ser. Lecture Notes in Computer Science, Y. Luo, Ed. Berlin, Heidelberg: Springer, 2013, pp. 101–110.

[34] J. J. López-Fernández, J. S. Cuadrado, E. Guerra, and J. de Lara, "Example-driven meta-model development," *Software & Systems Modeling*, vol. 14, no. 4, pp. 1323–1347, Oct. 2015.

[35] V. Zaytsev, "Language Design with Intent," in *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Sep. 2017, pp. 45–52.

[36] M. Fowler, "Language workbenches: The killer-app for domain specific languages," 2005.

[37] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning, "The State of the Art in Language Workbenches," in *Software Language Engineering*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, M. Erwig, R. F. Paige, and E. Van Wyk, Eds. Cham: Springer International Publishing, 2013, vol. 8225, pp. 197–217.

[38] A. Iung, J. Carbonell, L. Marchezan, E. Rodrigues, M. Bernardino, F. P. Basso, and B. Medeiros, "Systematic mapping study on domain-specific language development tools," *Empirical Software Engineering*, vol. 25, no. 5, pp. 4205–4249, Sep. 2020. [Online]. Available: https://link.springer.com/10.1007/s10664-020-09872-1

[39] M. Coblenz, G. Kambhatla, P. Koronkevich, J. L. Wise, C. Barnaby, J. Sunshine, J. Aldrich, and B. A. Myers, "PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design," *arXiv:1912.04719 [cs]*, Aug. 2020.

[40] J. D. Gould, J. Conti, and T. Hovanyecz, "Composing Letters with a Simulated Listening Typewriter," *Proceedings of the Human Factors Society Annual Meeting*, vol. 25, no. 1, pp. 505–508, Oct. 1981, publisher: SAGE Publications.

[41] A. Blackwell, M. Burnett, and S. Jones, "Champagne Prototyping: A Research Technique for Early Evaluation of Complex End-User Programming Systems," in *2004 IEEE Symposium on Visual Languages - Human Centric Computing.* Rome: IEEE, 2004, pp. 47–54.

[42] I. Poltronieri Rodrigues, M. de Borba Campos, and A. F. Zorzo, "Usability Evaluation of Domain-Specific Languages: A Systematic Literature Review," in *Human-Computer Interaction. User Interface Design, Development and Multimodality*, ser. Lecture Notes in Computer Science, M. Kurosu, Ed. Cham: Springer International Publishing, 2017, pp. 522–534.

[43] I. Poltronieri, A. C. Pedroso, A. F. Zorzo, M. Bernardino, and M. de Borba Campos, "Is Usability Evaluation of DSL Still a Trending Topic?" in *Human-Computer Interaction. Theory, Methods and Tools*, M. Kurosu, Ed. Cham: Springer International Publishing, 2021, vol. 12762, pp. 299–317, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-030-78462-1_23

[44] R. Halvorsrud, C. Boletsis, and E. Garcia-Ceja, "Designing a Modeling Language for Customer Journeys: Lessons Learned from User Involvement," in *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Oct. 2021, pp. 239–249.

[45] A. Barišić, J. Cambeiro, V. Amaral, M. Goulão, and T. Mota, "Leveraging teenagers feedback in the development of a domain-specific language: the case of programming low-cost robots," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing.* Pau France: ACM, Apr. 2018, pp. 1221–1229. [Online]. Available: https://dl.acm.org/doi/10.1145/3167132.3167264

[46] A. Stefik and S. Siebert, "An Empirical Investigation into Programming Language Syntax," *ACM Transactions on Computing Education*, vol. 13, no. 4, pp. 19:1–19:40, Nov. 2013.

[47] P. M. Uesbeck, A. Stefik, S. Hanenberg, J. Pedersen, and P. Daleiden, "An empirical study on the impact of C++ lambdas and programmer experience," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. Austin, Texas: Association for Computing Machinery, May 2016, pp. 760–771. [Online]. Available: https://doi.org/10.1145/2884781.2884849

[48] H. S. Borum, H. Niss, and P. Sestoft, "On Designing Applied DSLs for Non-programming Experts in Evolving Domains," in *Proceedings of the 24th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '21. Virtual Event, Fukuoka: Association for Computing Machinery, Oct. 2021.

[49] H. S. Borum, C. Seidl, and P. Sestoft, "Co-designing DSL quality assurance measures for and with non-programming experts," in *Proceedings of the 18th ACM SIGPLAN International Workshop on Domain-Specific Modeling*, 2021, pp. 31–40.

[50] M. van Amstel, M. van den Brand, and L. Engelen, "An exercise in iterative domain-specific language design?" in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE) on - IWPSE-EVOL '10*. Antwerp, Belgium: ACM Press, 2010, p. 48. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1862372.1862386

[51] M. Karaila, "Evolution of a Domain Specific Language and its engineering environment - Lehman's laws revisited," p. 7, 2009.

[52] M. Lehman, J. Ramil, P. Wernick, D. Perry, and W. Turski, "Metrics and laws of software evolution-the nineties view," in *Proceedings Fourth International Software Metrics Symposium*, Nov. 1997, pp. 20–32.

[53] A. Blackwell, "First steps in programming: a rationale for attention investment models," in *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, Sep. 2002, pp. 2–10.

[54] J. G. Redish, "Expanding usability testing to evaluate complex systems," *Journal of usability studies*, vol. 2, no. 3, pp. 102–111, 2007, publisher: Citeseer.

[55] F. Tomassetti and V. Zaytsev, "Reflections on the Lack of Adoption of Domain Specific Languages." in *STAF Workshops*, 2020, pp. 85–94.

[56] I. Poltronieri, A. F. Zorzo, M. Bernardino, and M. de Borba Campos, "Usa-DSL: usability evaluation framework for domain-specific languages," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18. Pau, France: Association for Computing Machinery, Apr. 2018, pp. 2013–2021. [Online]. Available: https://doi.org/10.1145/3167132.3167348

[57] M. Voelter, B. Kolb, K. Birken, F. Tomassetti, P. Alff, L. Wiart, A. Wortmann, and A. Nordmann, "Using language workbenches and domain-specific languages for safety-critical software development," *Software & Systems Modeling*, vol. 18, no. 4, pp. 2507–2530, Aug. 2019. [Online]. Available: https://doi.org/10.1007/s10270-018-0679-0

[58] R. Lämmel, *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer International Publishing, 2018.

[59] E. Aertbeliën and J. De Schutter, "eTaSL/eTC: A constraint-based task specification language and robot controller using expression graphs," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sep. 2014, pp. 1540–1546, iSSN: 2153-0866.

[60] S. Peyton Jones, J.-M. Eber, and J. Seward, "Composing contracts: an adventure in financial engineering (functional pearl)," *ACM SIGPLAN Notices*, vol. 35, no. 9, pp. 280–292, Sep. 2000.

[61] S. Frankau, D. Spinellis, N. Nassuphis, and C. Burgard, "Commercial uses: Going functional on exotic trades," *Journal of Functional Programming*, vol. 19, no. 1, pp. 27–45, Jan. 2009.

[62] J. van den Bos and T. van der Storm, "A Case Study in Evidence-Based DSL Evolution," in *Modelling Foundations and Applications*, ser. Lecture Notes in Computer Science, P. Van Gorp, T. Ritter, and L. M. Rose, Eds. Berlin, Heidelberg: Springer, 2013, pp. 207–219.

[63] M. Schuts and J. Hooman, "Industrial Application of Domain Specific Languages Combined with Formal Techniques," in *Proceedings of the 1st International Workshop on Real World Domain Specific Languages - RWDSL '16*. Barcelona, Spain: ACM Press, 2016, pp. 1–8. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2889420.2889421

[64] H. S. Borum and C. Seidl, "Survey of Established Practices in the Life Cycle of Domain-Specific Languages," 2022, submitted to: MODELS'22 (TODO: update).

[65] "Official page for Language Server Protocol." [Online]. Available: https://microsoft.github.io/language-server-protocol/

[66] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ser. ICFP '00. New York, NY, USA: Association for Computing Machinery, Sep. 2000, pp. 268–279.

[67] "Official page for Debug Adapter Protocol." [Online]. Available: https://microsoft.github.io/debug-adapter-protocol/

[68] O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich, "Code generation from a domain-specific language for C-based HLS of hardware accelerators," in *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct. 2014, pp. 1–10.

[69] J. A. Barriga, P. J. Clemente, E. Sosa-Sánchez, and l. E. Prieto, "SimulateIoT: Domain Specific Language to Design, Code Generation and Execute IoT Simulation Environments," *IEEE Access*, vol. 9, pp. 92 531–92 552, 2021, conference Name: IEEE Access.

[70] P. Wadler, "Deforestation: transforming programs to eliminate trees," *Theoretical Computer Science*, vol. 73, no. 2, pp. 231–248, Jan. 1988. [Online]. Available: https://doi.org/10.1016/0304-3975(90)90147-A

[71] A. Gill, J. Launchbury, and S. L. Peyton Jones, "A short cut to deforestation," in *Proceedings of the conference on Functional programming languages and computer architecture - FPCA '93*. Copenhagen, Denmark: ACM Press, 1993, pp. 223–232. [Online]. Available: http://portal.acm.org/citation.cfm?doid=165180.165214

[72] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[73] H. S. Borum and M. T. Clausen, "Generating Efficient Management Actions for the Pension Industry," 2022, in Review.

[74] "BenchmarkDotNet," 2021, https://benchmarkdotnet.org/. Accessed Dec 2021. [Online]. Available: https://benchmarkdotnet.org/

[75] S. Dragule, S. G. Gonzalo, T. Berger, and P. Pelliccione, "Languages for Specifying Missions of Robotic Applications," in *Software Engineering for Robotics*, A. Cavalcanti, B. Dongol, R. Hierons, J. Timmis, and J. Woodcock, Eds. Cham: Springer International Publishing, 2021, pp. 377–411. [Online]. Available: https://doi.org/10.1007/978-3-030-66494-7_12

[76] N. Kapre and S. Bayliss, "Survey of domain-specific languages for FPGA computing," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2016, pp. 1–12, iSSN: 1946-1488.

[77] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography," *ACM SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, Jun. 2000. [Online]. Available: https://doi.org/10.1145/352029.352035

[78] T. T. Hildebrandt and R. R. Mukkamala, "Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs," *Electronic Proceedings in Theoretical Computer Science*, vol. 69, pp. 59–73, Oct. 2011, arXiv:1110.4161 [cs]. [Online]. Available: http://arxiv.org/abs/1110.4161

[79] M. F. Madsen, M. Gaub, T. Høgnason, M. E. Kirkbro, T. Slaats, and S. Debois, "Collaboration among adversaries: distributed workflow execution on a blockchain," in *Symposium on Foundations and Applications of Blockchain*, 2018, p. 8.

[80] H. S. Borum, M. F. Madsen, and S. Debois, "Static Secrecy Guarantees For DCR Graphs," 2022, work in progress (Todo Update).

[81] R. Fagin, J. Y. Halpern, Y. Moses, and M. Vardi, *Reasoning About Knowledge*, Jan. 2004. [Online]. Available: https://direct.mit.edu/books/book/1825/Reasoning-About-Knowledge

[82] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977. [Online]. Available: https://doi.org/10.1145/359636.359712

[83] "JavaScript With Syntax For Types." library Catalog: www.typescriptlang.org. [Online]. Available: https://www.typescriptlang.org/

[84] G. Bierman, M. Abadi, and M. Torgersen, "Understanding TypeScript," in *ECOOP 2014 – Object-Oriented Programming*, ser. Lecture Notes in Computer Science, R. Jones, Ed. Berlin, Heidelberg: Springer, 2014, pp. 257–281.

[85] "HackerRank 2020 Developer Skills Report." [Online]. Available: https://info.hackerrank.com/rs/487-WAY-049/images/HackerRank-2020-Developer-Skills-Report.pdf

[86] "Stack Overflow Developer Survey 2021." [Online]. Available: https://insights.stackoverflow.com/survey/2021

[87] "PYPL PopularitY of Programming Language index," library Catalog: pypl.github.io. [Online]. Available: https://pypl.github.io/PYPL.html

[88] M. S. Alvim, K. Chatzikokolakis, A. McIver, C. Morgan, C. Palamidessi, and G. Smith, *The Science of Quantitative Information Flow*, ser. Information Security and Cryptography. Cham, Switzerland: Springer, Springer Nature, 2020.

[89] S. Mathôt, D. Schreij, and J. Theeuwes, "OpenSesame: An open-source, graphical experiment builder for the social sciences," *Behavior Research Methods*, vol. 44, no. 2, pp. 314–324, Jun. 2012. [Online]. Available: https://doi.org/10.3758/s13428-011-0168-7

[90] "iMotions: Unpack Human Behavior," library Catalog: imotions.com. [Online]. Available: https://imotions.com/

Appendix A

On Designing Applied DSLs for Non-Programming Experts in Evolving Domains

On Designing Applied DSLs for Non-programming Experts in Evolving Domains

Holger Stadel Borum
Department of Computer Science
IT University of Copenhagen
Copenhagen, Denmark
hstb@itu.dk

Henning Niss
Edlund A/S
Copenhagen, Denmark
henning.niss@edlund.dk

Peter Sestoft
Department of Computer Science
IT University of Copenhagen
Copenhagen, Denmark
sestoft@itu.dk

***Abstract*—Domain-specific languages (DSLs) have emerged as a plausible way for non-programming experts to efficiently express their domain knowledge. Recent DSL research has taken a technical perspective on how and why to create DSLs, resulting in a wealth of innovative tools, frameworks and technical approaches. Less attention has been paid to the design process. Namely, how can it ensure that the created DSL realises the expected benefits? This paper seeks to answer this question when designing DSLs for highly specialised domains subject to resource constraints, an evolving application domain, and scarce user participation. We propose an iteration of alternating activities in a human-centred design method that seeks to minimise the need for expensive implementation and user involvement. The method moves from a low-validity exploration of highly diverse language designs towards a higher-validity exploration of more homogeneous designs. We give an in-depth case study of designing an actuarial DSL called MAL, or Management Action Language, which allows actuaries to model so-called future management actions in asset/liability projections in life insurance and pension companies. The proposed human-centred design method was synthesised from this case study, where we found it useful for iteratively identifying and removing usability problems during the design.**

***Index Terms*—Model-driven engineering, Domain-specific language, Human-centred design**

I. Introduction

As computations have become prevalent in most parts of society, many domain experts with little programming experience (*non-programming experts*) are required to input their knowledge into complex computer systems as part of their everyday lives. Domain-specific languages (DSLs) promise a way for experts to do so without relying on the assistance of programming professionals. By tailoring a language to the needs of domain experts, a DSL may be more flexible and expressive than a traditional graphical user interface (GUI) while still being easier to use than a general-purpose programming language (GPL).

In this paper, we are concerned with how to tailor a language to the needs of non-programming domain experts. In other words, we investigate how to ensure that a DSL realises its expected benefits when one sets out to design it to accommodate non-programming experts. It is more difficult for a language designer to anticipate how non-programming users approach a language than how programming professionals do. Therefore, we seek to answer the question by proposing a design method based on our experiences using human-centred design (HCD) to structure the language design process for a DSL called MAL. We distinguish between a method as a general design framework and a technique as a concrete tool that can be applied as part of a method. MAL was designed in a collaboration between Edlund A/S and the IT University of Copenhagen to allow Edlund's customers to model company-specific management actions on a general asset/liability projection platform (see Section VII-A). Based on our experiences with designing MAL, we propose a two-phase design process moving from low-validity exploration of diverse language designs towards a higher-validity exploration of less diverse designs. We seek to mitigate the risk of designing a language unusable by non-programming experts by having an HCD approach that incorporates the user perspective into the design process through observation and continuous evaluation.

We consider DSLs as a specific model-driven engineering (MDE) practice where the abstract syntax of a DSL is a metamodel, language artefacts are models, and code generations are transformations. Due to the many usages of DSLs, we limit our investigation to consider only the design of *applied* DSLs (ADSLs) for non-programming experts. An ADSL is a DSL that seeks to alleviate problems in large, complex software systems as opposed to exploring more theoretical aspects of programming languages. While there are other names for similar types of DSLs (*application domain* DSL [1] or *real-world* DSL [2]), we prefer to use the term *applied* since all DSLs exist within the *real world* and an *application domain*. An ADSL lives within a complex software ecosystem with its corresponding business processes. An ADSL must be compatible with this context. At the same time, the introduction of an ADSL often aims to change business processes by empowering the end-user with new capabilities and corresponding responsibilities. Therefore, an ADSL is likely to evolve to remain compatible with its context, which it intentionally sets out to change.

Innovation Fund Denmark (7076-00029B)

The paper will progress as follows: Section II describes the method of the paper. Section III describes state of the art. Section IV presents a language classification based on evolution relevant to the design of ADSLs. Section V identifies challenges to designing ADSLs for non-programming experts. Section VI describes our proposed iterative, two-phased design method. Section VII presents our in-depth case study of creating MAL leading to the proposed method. Finally, Section VIII discusses threats to the validity of our case study.

The contributions of this paper are:

- A classification and discussion of language evolution based on Manny Lehman's program classification.
- An identification of ADSLs' characteristics and design challenges.
- A proposal of a design method for creating ADSLs.
- A case study of MAL's design process leading to the proposed method.

II. Method

In this paper, we propose a design method for ADSLs based on our experiences with designing MAL. We do so from the position that a detailed qualitative case study is a valid scientific method for providing grounds for generalisations [3]. We use our in-depth understanding of challenges faced when designing MAL to synthesise a general approach to ADSL design. Although we use these experiences as a justification for the proposed method, they should only be thought of as a justification and not as an evaluation of the method. Additional experiments and experiences with using the method are required for evaluative purposes. For the sake of clarity, we choose to present the proposed method followed by the case study since it allows the reader to more easily follow the case study.

III. State of the art

Programming language usability has been a driving factor in the history of programing language design [4] [5] [6]. The movement from unsafe, low-level languages towards safe, high-level languages can be attributed to a demand for easier ways to construct large, complex, efficient, safe, and correct software systems. While the idea of using human-computer interaction (HCI) techniques in language design [7] has been applied before, there is little well-established design methodology ensuring the usability of a programming language. This is the case even though usability is a primary concern of programming language design. Instead, new programming paradigms and language features have claimed to improve the programming experience and have either succeeded or failed through industrial or academic adoption or lack thereof. For a language designer, this survival of the most popular approach to language improvements provides no guidance on how to design a language. Furthermore, although there exist many heuristic guidelines and rules for language design [8] [9], these do not help the designer choose concrete design activities. As an example, the theory of *Physics of Notation* (PoN) [10] contains nine guiding principles for designing visual notations but provides no design-procedural guidelines. Van der Linden and Hadar [11] find that few PoN practitioners report user involvement when eliciting requirements for a visual notation.

Human-centred design (HCD) seems like the most promising methodology for guiding a designer on how to create programming tools [12], including ADSLs for non-programming experts. HCD seeks to include the user in the design process to ensure that a designed artefact is innovative and solves the user's needs, i.e., it has high usability [13]. Preliminary research findings suggest that while expert programming language designers believe human-centred techniques are of great value to language design, very few use them in practice [14]. Roughly, HCD techniques can be divided into *formative techniques* that create new ideas and *evaluative techniques* that evaluate a specific idea. Many techniques have been adapted from general HCI techniques to the context of programming language design [15]. Evaluative techniques have a central position in HCD for programming languages, possibly because programing language design has historically neglected evaluative studies [16] [17]. For our proposed method, we group the evaluative techniques into the following two general cases:

High validity, low variability experiments, or randomised controlled trials, explore few independent variables with a large sample size. These statistically significant experiments may uncover fundamental truths about programming languages' usability with a rigour comparable to medical studies [16] [18]. For example, such work has involved the effect of static type systems [19] or choice of keywords [20]. From the perspective of ADSL design, such experiments are costly due to the required number of participants and time spent conducting the experiments. However, the results of such existing studies may be used as heuristic guidelines for future languages. This approach to design has been proposed as the *evidence-based programming language design* method, which seeks to base language design choices on relevant usability studies [21]. Although such a design method may eventually be of great value in the design of GPLs, such a method is less feasible to transfer to the design of a specific DSL. The primary problem is that the method requires an experiment that investigates a particular design decision. If we seek a domain-specific solution for a previously unexplored domain, it is unlikely that existing experiments directly address our problems.

Low validity, high variability experiments explore many different independent variables on a small sample size as input to design decisions. The idea is that "20 cheap studies of a variety of ideas and features are likely to be far more valuable than one expensive study of a particular feature", as stated by the *Champagne Prototyping* technique

[22]. These evaluative techniques are typically variants of a *Wizard of Oz* technique [23], which seeks to evaluate a system on participants by using an approximation of the final system[1]. *Champagne Prototyping* is used to validate end-user programming features by exposing users to scenarios in their normal rich functioning system while only mimicking the core feature of interest. The *PLIERS* method, which will be discussed later, also proposes a *Wizard of Oz* technique where the language designer functions as a type system and compiler that can provide users with the illusion of interacting with the final system. Although these techniques provide a way of exploring a diverse set of design ideas, they do so at the cost of the validity of their findings.

HCD methods for DSL design are more difficult to find. So far, we have discussed only human-centred techniques as parts of a language design process, except for the *evidence-based language design* method. Although these techniques are of great use and value to designers, they provide little guidance on how to structure a DSL design process. Even the guidance in the foundational book *Domain-Specific Languages* [25] does not feel comfortable making more specific suggestions than to "[t]ry out different ideas on your target audience". However, some practitioners do provide some guidance. Although early work descriptively divided DSL development into the phases of *decision*, *analysis*, *implementation*, and *deployment* [26], there is seemingly a contemporary consensus on using an iterative approach [1] [12]. The book *DSL Engineering* [1] recommends early, agile development of a core language and classifies language development according to domain experts' knowledge. The more recent book *Software Languages* [27] only deals "superficially with domain analysis, language design, evolution, and retirement." The developers of FlowSL recommend early usability evaluation when developing DSLs for non-programmers [28]. Another case study shows how to evaluate a DSL by treating it as a traditional GUI [29]. The *Design Your Own Language* [30] toolkit consists of 96 cards, each representing a possible design aspects that affects the behaviour of users. While the toolkit thoroughly presents and categorises these decisions important to users, it lacks processual guidelines for practitioners who seek to navigate them. Inspired by *free and open software communities*, the DSL named *Colaboro* [31] may be used for decision making in community-based design projects. Lastly, users may be included at fixed points in the design process, for example, through a questionnaire to make syntax decisions [32].

For general-purpose programming languages, the *PLIERS* method makes specific suggestions for conducting human-centred, iterative language refinement [33]. It provides a general design process, a set of HCD formative and evaluative techniques, and mitigation for common problems for conducting such experiments. Although there is an overlap between DSL design and GPL refinement, a substantial part of the method does not apply to the design of ADSLs for non-programming experts. For example, the method assumes users with substantial programming experience and a relatively stable domain. It does so when it suggests that design questions on a specific language can be back-ported into a more commonly known language.

Analytical frameworks for understanding and analysing usability issues of programming languages are also used for evaluative purposes. They can be used as a theoretical foundation to guide an evaluative technique and to interpret its findings. Although the *Cognitive Dimension Framework* [34] was developed to identify usability issues in visual programming languages, its 14 analysis dimensions have proven useful for analysing many kinds of programming languages [33] [35] [36]. The *Attention Investment Model* is another framework that seeks to explain whether end-users will use a programming tool. It does so by weighing the pros of adopting the tool against the cons and risks of doing so [37]. From the perspective of ADSLs, the theory claims that users will only use an ADSL if the benefits of using it outweigh both problems and potential risks. Therefore, an ADSL should not only make a modest improvement of the current practices.

The technical design of DSLs comprises programming languages, tool support, and workbenches for DSL construction [38]. Several tools exist to alleviate the pains of developing DSLs by reducing the development cost. To mention some: MPS, Xtext, Racket, or the Language Server Protocol all ease the implementation of a DSL by allowing the reuse of IDE features, language features, and even complete languages. There are plenty of case studies and examples of how these tools can be used to create DSLs [39] [40] [41] [42]. Although the technical choices of DSL implementation affect a design method and vice versa, we will not consider technical design as part of our design method. This separation allows our method to stringently consider the human-centred perspective and practitioners to make their own technological decisions.

IV. Language evolution

In a classical paper from 1980, Manny Lehman proposed to classify programs into the three categories of *S*, *P*, and *E*-programs [43]. Briefly, an *S*-type program is precisely derivable from its *specification*, a *P*-type program may have a precise *problem* specification but requires an approximation in its implementation, and an *E*-type program is *embedded* in the real world and is thereby part of its own application domain. With the progression from *S*-type to *E*-type programs follows more software evolution caused by the distance between a program's specification and its actual implementation and usage.

[1] Some high validity experiments have a similar approach [24].

TABLE I
EXAMPLES OF LANGUAGES BELONGING TO THE DIFFERENT CLASSES.

Category	Languages
***S*-type**	λ-calculus, regular expressions, Communicating Sequential Processes
***P*-type**	C, Java, C#
***E*-type**	ADSLs, PHP, JavaScript, Python

Inspired by Lehman, we derive a similar classification for *programming languages* where each class has its own reasons for language evolution. We have found that this classification provides a good framework for distinguishing causes of language evolution, based on our knowledge of current and historical programming languages and their development over time. While *S*-type languages are very stable, *P*-type languages continuously evolve and even more so for *E*-type languages. Languages are in themselves interesting to consider in the perspective of evolution since languages often outlive software written in them. We will use this classification to argue that an ADSL is an *E*-type language that necessitates a design method that handles language evolution. Table I shows examples of languages belonging to different classes.

We distinguish between *exogenous* and *endogenous* evolutionary forces corresponding to Lehmann's first and sixth laws of evolution [43]. The exogenous forces come from changes in the domain itself (caused by new legislation, new business practices, and the like) and from changes in the underlying technology (such as the shift from mainframes to stand-alone desktop computers to networked desktops to web servers, etc.). The endogenous forces come from users adopting the language, liking some aspects of it and disliking or missing others, causing them to request changes in the language.

An *S*-language is created for the sake of its own specification that serves as a cardinal example of an external concept. An *S*-type language is designed to demonstrate or model the external concept, which means that the primary reason for writing a program in the language is to demonstrate the external concept's properties. An *S*-type language only evolves when its external concept changes or if a better model is discovered. Therefore, *S*-languages are very stable, and it is uncommon for an *S*-language to evolve. In fact, an *S*-language will likely evolve into a new, independent language, demonstrating some other external concept. Many languages designed for programming language research and education are *S*-languages. The λ-calculus is an *S*-language since its purpose is to model computations with its semantics being of greater interest than its application. The class also contains other languages such as a simple imperative C-like language used to teach compiler construction or simple state machine-based languages [25] used to demonstrate DSLs as a concept. While these languages may resemble *P*-type languages, they are *S*-type since they are reduced to fundamental concepts directly aligning them with their specification.

A *P*-language is created for the sake of its programs and their application context. A *P*-language is designed to create and maintain complex programs, which means that it considers software engineering aspects such as hardware, efficiency, code-reuse, and broader development context. A *P*-language will primarily evolve in response to exogenous forces in its application context: new hardware technologies, software engineering practices, or programming language paradigms. *P*-languages are more concerned with language usage within a specific computational model than the language itself. As a result, there are many pairs of *S* and *P*-type languages where an *S*-type language demonstrates the computational model while a *P*-type language makes the model practically usable. Some examples of pairs are λ-calculus & Haskell, Algol-60 [44] & Algol-W [45] or Pascal [46], regular expressions & Perl compatible regular expressions [47] and Communicating Sequential Processes [48] & Occam [49] or Erlang [50].

An *E*-language is created for the sake of its programs that, in turn, are shaped by their application context. Like *P*-languages, an *E*-language is designed to create and maintain complex programs, but where a *P*-language derives its form from some computational model, an *E*-language is primarily influenced by its application context. An *E*-language is likely to evolve due to both exogenous and endogenous forces. This means that the development of many *E*-languages has an ad-hoc feeling compared to *P*-languages. PHP is an example of an *E*-language created for the purpose of dynamic server-side creation of webpages with database access. As web technology evolved, so did PHP with the additional user requirements, security fixes, and larger web frameworks. Also, many DSLs are *E*-languages since they are shaped by their domain, which is part of the application context.

Some properties of this classification are best made explicit: First, a language is not inherently *S*-type, *P*-type, or *E*-type. Instead, it is the purpose and usage of a language that determines its class. Thus, the class of a language may change over time. Second, there is a clear correlation between a *P*-type or *E*-type and whether it is a GPL or a DSL. However, the classes are not equivalent. For example, a stable DSL that serves as a top-level state-machine interface is a *P*-type language since it makes a computational model readily available to its users. Likewise, we have already argued why a GPL such as PHP has many *E*-type characteristics. Third, the classification does not establish that *S*-type languages are superior to *E*-type languages or vice versa. Neither is the point that one should write *S*-type programs in an *S*-type language and so forth. The point of the classification is to provide a new perspective for the reasons for language evolution. Based on the above classification, we have the following hypotheses:

Hypothesis 1: Most ADSLs are *E*-type languages and part of an endogenous evolutionary cycle.

Such an evolutionary cycle would unfold as follows: an initial version of the DSL supports only parts of the domain (to avoid wasted implementation effort) and additionally contains misunderstandings of the domain. Nevertheless, the implemented parts may be successful in attracting users who request additions and adaptations, which causes the DSL to subsequently attract more users who request new features.

Hypothesis 2: Language evolution is hindered by a formal language specification, which means that a formal language specification in itself moves a language towards an *S*-type language.

Evolving a formal language specification is costly, especially if the specification guarantees properties such as type safety. Therefore, a language creator may, rightfully, opt not to evolve a language when the evolution requires its formal specification to change. When this happens, the language specification is in itself deemed more important than the language usage, which is a characteristic of *S*-type languages. This may be the reason why Standard ML is more *S*-like than similar languages such as OCaml and F#.

We use the hypotheses to make the following assertion: a design method for ADSLs must handle the described endogenous evolutionary cycle. For this purpose, it may be counterproductive to spend time formally describing the developed language and proving language properties. Formalisation may be desirable for other reasons, but it may impede the language design process, at least in the early stages.

V. Design challenges

In this section, we describe the challenges of having a human-centred design process for creating ADSLs for non-programming experts. Most of these challenges stem from seeking to have users evaluate language designs, as such evaluation requires some way for users and designers to communicate complex concepts.

Challenge 1: High impact systems

ADSLs may have a high impact in the sense that they model important phenomena so that errors can have serious consequences. Whether domain experts use the DSL to estimate a pension company's solvency [51], model financial contracts [52] [53], or manage telecommunication switches [54], mistakes can come at a high price. A design process for a high impact ADSL should consider ways to minimise the risk of errors.

Challenge 2: Few experts

A domain may have few experts, even if it is of great importance. In general, the more specialised a domain becomes, the fewer experts there are in the domain. In some highly specialised domains, such as asset/liability projections of Danish pension companies, there are very few experts, meaning that a design method cannot rely on many repeated user evaluations with domain experts. This scarcity challenges the core of HCD since users are vital for the method. For our purposes, we will assume that the design team always has one expert available since we deem it unlikely that one would create a DSL for a specialised domain without an expert's help.

Challenge 3: Gap between designer and user

There will always be a knowledge gap between designer and user. This gap is vast in the design of an ADSL since both language designers and domain experts come from very specialised domains which require years of experience and education to grasp. Even though neither person will fully comprehend the other's domain, a design method must seek to bridge this gap since any form of miscommunication may translate into problems with the design.

Challenge 4: Evolving and amorphous domains

As stated in section IV, an ADSL will likely evolve, even during its development. Domain experts may discover new opportunities and requirements during the design process, which means there is a risk of a language being outdated before it is completed. Therefore, the design method should allow for domain changes even late in the process.

When there are substantial uncertainties (such as unfinished mathematical models or unclear legislation) in a domain, we call it *amorphous*. By this, we mean that domain experts are actively working on clarifying these uncertainties resulting in an evolving domain. These uncertainties make it difficult to design a DSL since they move the design in the direction of general-purpose solutions. An amorphous domain leads to a particular form of evolution, resulting in a decrease in domain entropy. While one should seek to answer all domain questions, it may not be possible to do during the design phase.

Challenge 5: Intangible and abstract product

A domain-specific language is an intangible and abstract product that makes it difficult for users and customers to monitor it during development. This challenge applies to all software products [55] but even more so for DSLs because it can be hard to present a DSL to users. Presenting a DSL's grammar and semantics is likely too abstract for the user, while presenting a single program may be too concrete to demonstrate broad design implications.

Challenge 6: Part of a complex system

Since an ADSL is created as part of a complex software system, the DSL may seek to replace a significant codebase. When this is the case, users may only be interested in complex models corresponding to thousands of lines of code. This means that it may not be possible to provide users with a small core DSL, which will be incrementally expanded and improved.

VI. Proposed method

We propose a pragmatic, iterative, two-phased method to design ADSLs for non-programming experts. This method is synthesised from our experience with designing MAL, which will be discussed in Section VII. With the proposed method, we strive for a process that lets practitioners create DSLs that realise their expected benefits while handling the challenges described in the previous section. This approach is opposed to striving for a design process that leads to an optimal language where a statistically significant experiment justifies each design decision. Such a method should be applicable to anyone from a novice to an expert DSL designer but customisable to the needs of a specific design context. By proposing this method, we seek to contribute to a design methodological discussion of how such a method can help practitioners tailor a language to a domain.

We seek to use an HCD approach because the method promises a way of exploring innovative ideas while ensuring that these ideas are grounded in users' needs. At its core, HCD is an iterative process that consists of two activities. The first activity creates an artefact or a prototype; the second activity evaluates the prototype with a user experiment. This process allows us to continuously evaluate and evolve a language design, but it immediately raises two questions. How do we create an artefact suitable for evaluation? and how is said artefact evaluated?

Our two-phase design method moves from a low-validity exploration of diverse prototypes *towards* a high-validity exploration of few prototypes[2]. The phase of low-validity exploration consists of small, fast design iterations using pseudocode prototypes allowing the designer to explore the language domain through vastly different language designs. During this phase, an in-team expert serves as the best approximation of actual users. When the prototypes of the low-validity exploration converge, the second phase of design validation begins. In this second phase of design validation, the loosely evaluated design from the first phase is implemented and tested to find usability problems.

A. Low-validity exploration

The first phase consists of small design iterations that seek to explore the design space at a low cost. Cheap and flexible prototypes are necessary for this process since the time used on a prototype is inversely proportional to the number of iterations. In this phase, prototypes should be purely textual and could be called pseudocode prototypes analogous to paper prototypes. These prototypes should be supplemented by formative techniques such as corpus analysis, interviews, natural programming, or domain-driven design [12] [56]. During these activities, one should be looking for desired language properties and constraints to incorporate into the design as early as possible.

The purpose of a prototype is to investigate a specific design decision and facilitate a discussion on the decision. Therefore, a prototype should be focused on exploring the solution to only a few problems, so it gives information on the subject of interest. It may do so in extreme, even unrealistic ways as long as it serves as an idealised example and not as an end product. Also, a prototype may tackle macro-level questions such as how to improve overall program comprehension. In that sense, the prototyping process uses the hermeneutic circle since it views the part in the context of the whole and vice versa.

The in-team expert is used to evaluate each prototype. The goal is to prompt the expert for questions and opinions such as, "Why can I not just do X?", "What if I want to do Y?", or "What is the purpose of Z?". However, merely showing a pseudocode prototype may not give the expert sufficient grounds to provide feedback. Therefore, part of the evaluation should be to walk through different scenarios in a text editor to show how one would work with the prototype. A scenario could be to change a specific business rule, write a new rule from scratch, or find an error in the program. When possible, the expert should dictate what to do or write. Again this is analogous to what one would do with a paper prototype of a graphical user interface. During the phase of low-validity exploration, the designer will experience that the language design of the prototypes become more and more stable. As this happens and the number of unexplored, potentially viable designs also diminishes, we say that the prototypes converge to a single language design. In other words, a consensus should emerge on how one would like to express different computations. When this consensus is reached, the phase of low-validity exploration ends.

B. Design validation

The low-validity exploration produces a rough language design that neither the designer nor the expert objects to. There is, however, still a risk that the language design contains significant flaws. First, the design may have unrealistic assumptions on the possible language guarantees since the design builds upon pseudocode prototypes. Second, the low-validity exploration may have biased the in-team expert, or they may, for other reasons, not represent domain experts in general. Phase two seeks to mitigate these risks by creating a lightweight language implementation and testing it with as many participants as feasible[3]. This process should ensure that the language design is based on realistic assumptions and usable by outside experts. Any problems discovered should be addressed by changing the design leading to, if

[2]With an emphasis on towards since we are not proposing experiments which give statistically significant results.

[3]The number of needed usability tests has been discussed at least since Nielsen's and Launder's mathematical analysis of usability problems [57]. From our perspective, it is unlikely that someone ends up with the possibility of performing too many usability tests when creating ADSLs.

TABLE II
CHALLENGES AND REMEDIES

Challenge (section V)	Remedy
High impact systems	No specific
Very few users	Initially use in team expert, then validate with external users
Designer and expert gap	Small and fast iterations
Intangible product	Text-editor prototypes and demonstrations
Evolving domain	Incremental and flexible approach
No small programs	No expectation of small programs

possible, new tests. These potential problems are the reason the implementation needs to be as lightweight as possible. There are two products of the second phase: first, a lightweight language implementation ready to be put to use and second, knowledge of the language's strengths and usability issues.

C. Tackling challenges

Here we will address why the proposed method handles the challenges identified in Section V. These are summarised in Table II. The small and fast design iterations facilitated by pseudocode prototypes serve a multitude of purposes. First, they seek to bridge the gap between the designer and the in-team expert by letting the designer become familiar with the domain and the expert to become familiar with DSL concepts. Second, they allow for flexibility in the design process to handle domain discoveries made by domain experts and the corresponding evolution of the DSL. Third, the repeated demonstrations of prototypes try to lessen the intangibility of the system. The challenge of having few test participants is handled by using the in-team expert as a best approximation of users. The second phase primarily exists to mitigate risks introduced by the first phase, namely that the prototype may be unrealistic and that real users may be different from the in-team expert. There is no specific mitigation for developing a high-impact DSL apart from improving usability and incorporating domain constraints into the DSL, thereby eliminating some potential user errors.

VII. CASE STUDY

In this section, we describe a case study on our experience using the design method to design the actuarial DSL, Management Action Language (MAL). It would be misleading to say that we had a fixed two-phase design methodology from the outset of the project. Instead, the two-phased method was discovered and synthesised during the project to account for the identified risks. This process introduces a threat to the validity of our findings (discussed in Section VIII), but recognising the threat is the first part of mitigation.

Although we played the active part of designers in the process, we take the perspective of process observers in this section. For the sake of readability, we will call the in-team expert Erin and the designer David. First, we give a detailed description of our design context and why MAL serves as a cardinal example for an ADSL. Then we discuss our experiences using the two-phase method. Finally, we discuss the methodological problems experienced throughout the project.

A. Design context

MAL was created in cooperation between the Danish software company Edlund A/S, specialising in software for the life insurance and pension industry, and the IT University of Copenhagen. One of Edlund's products is a platform that projects the asset/liability balance of a pension company in accordance with financial regulations. This projection of assets and liabilities is used to ensure and document that a pension company will remain solvent in the future. On this platform, a company must model its business rules, so-called management actions. MAL provides companies with a way of doing so.

MAL seeks to alleviate the following pains in the projection platform:

- There is a high entry barrier for actuaries for modelling and understanding business rules in a GPL.
- Some domain properties are difficult to ensure in a GPL.
- It is a security risk to allow actuaries to model and execute models written in a GPL.
- Customers are provided with a template GPL program where it is difficult to pick and choose different management actions.
- It is difficult for Edlund to experiment with performance initiatives applicable to customer models expressed in a GPL.

Given enough time and training, there is no doubt that actuaries could learn to use any language. Therefore, MAL more ambitiously aimed to make the language enjoyable to its users, which could be a selling point of the projection platform.

MAL is a cardinal example of an ADSL with all of its challenges (see Section V). It resides within an advanced software platform and a business-customer relation where Edlund must be competitive. To provide users with language flexibility, MAL generates valid GPL programs and allows invoking some external GPL code. There are few potential users of the language, each interested in complex models of management actions of a company, including policies, reserves, assets, cash flows, future discretionary benefits, etc. The domain of MAL is evolving and amorphous. The amorphicity stems from uncertainties in the exact requirements of the Danish FSA and ongoing actuarial research into the mathematics of asset/liability projections [58] [59]. In addition, the Danish pension industry manages assets corresponded to 300% of Denmark's GDP in 2019, which means that mistakes made in such projections could have severe consequences for the Danish economy [60].

B. Execution

We will now describe how the two phases were executed to create MAL. We first describe what happened during the phase and how the method helped us to overcome problems. This description is followed by a list of condensed lessons related to the phase.

1) Low-validity exploration: The phase of low-validity exploration was used to explore a wide range of language designs and ideas by iterating through low-cost pseudocode prototypes. Corpus analysis, interviewing, and domain modelling were the primary formative techniques used to create these prototypes. The corpus analysis consisted of analysing existing GPL programs that MAL was to replace. While this analysis allowed for a thorough domain investigation, it also hindered using *natural programming* as a technique since the in-team expert, Erin, could simply point to existing code, when asked how to express some computation. Collaborative domain modelling served as a better technique to get prescriptive input on how Erin wanted to work in the domain. During the project, David observed that several ideas from this collaborative domain modelling showed up in the GPL programs.

Several ideas were rejected in their initial form but modified and included in later prototypes. For example, early in the process, David recognised that the DSL needed some way for actuaries to model a new quantity that they wanted to compute. One of the first prototypes explored the possibility of inferring a data model from a written program. The idea was that actuaries could simply calculate and use a quantity by assuming it existed. Although the lack of explicit modelling turned out to be a bad idea, further prototype refinement led to a language where usability tests indicate that users enjoy modelling data in MAL. Later, David identified the problem that programs became bloated with iteration constructs. Again, David created a prototype that explored the possibility of having implicit iterations. This also turned out to be an unusable idea since its extreme way of making programs less verbose led to incomprehensible programs. However, this idea later reappeared as projections on the level of portfolios, which users generally like. The point of describing these iterations is to show how they facilitated the exploration of extreme ideas, which, due to their low cost, could be discarded or refined as David saw fit. An experienced language designer could likely have avoided some of the, in hindsight, design missteps, but from this method's perspective, that would only mean that an experienced designer needs fewer design iterations.

Erin, who was developing the mathematics to be implemented in MAL, was used as the best approximation of users during the design process. Therefore, it was difficult for Erin to state exact requirements and limitations for the language since she could only say how things looked right now and in the near future. Often, it was just "possible" that some functionality was required or "not likely" to be needed. Instead of requiring Erin to scope the domain, we primarily analysed her existing computations to synthesise a language design. This approach had the benefit that Erin could compare existing computations with equivalent DSL solutions, which gave her a better basis for questioning design choices.

After approximately ten prototypes, David had a rough idea of how the language would look. He had identified important functionality of the language (data modelling, calculations, and output specification) and had a sketch of a language that provided said functionality. However, there were still unresolved questions. First, since Erin had been used as an approximation of users, it was still unclear whether other actuaries would actually enjoy using the language. Also, in the initial language sketch, much attention was paid to how programs of interest could be modelled. Less attention had been paid to how a program would fit into Edlund's customer relationship, where the company provides template solutions to several customers.

Lesson 1: Do not be afraid of seemingly stupid, crazy, or unrealistic ideas. Even if an idea does not end up in the final language, it can still shape and delimit the language.

Lesson 2: Do not require the domain expert to make absolute statements about functionality. Instead, ask how likely it is that some functionality is required. Design the language for functionality, which has a high probability of being required.

Lesson 3: A comparison between existing models and equivalent DSL models will likely prompt a reaction from domain experts. So will modifying models expressed by the DSL.

Lesson 4: For a novice designer, it is easy to come up with unrealistic ideas. Therefore, a novice designer should seek to validate that an idea is realistic.

2) Design validation: In the second phase, David sought to validate that the language was highly usable by actuaries. The plan was to conduct traditional usability experiments with actuaries to find and weed out usability issues. Therefore, phase two began with a lightweight implementation of the DSL and an identification of important usability goals. The Cognitive Dimension Framework [34] was used to identify these goals and corresponding tasks. For example, one goal was that "the user should understand the different kinds of data and where the data comes from" (hidden dependencies).

In total, two usability tests were conducted; one with an Edlund actuary and one with a customer actuary. The test consisted of training (30 min), task solving (120 min), and a semi-structured interview (30 min). The tests strengthened

David's belief in many design choices since both users saw potential in MAL's data modelling and were able to understand and modify complex programs. However, the usability tests also found significant problems with syntactical choices, training material, and error messages. An example of one of these problems was that the prototype had moved from a C-like towards an ML-like syntax without much complaint from Erin. When exposing fresh actuaries to the language, it became clear that actuaries are more experienced with a C-like notation. As one participant explicitly stated: "[they] were missing curly braces and semicolons for structure". Although such a syntactic problem is easy to fix, it is essential to identify to flatten the learning curve of the DSL.

During the second phase, it became clear that the language should not only be tailored to its domain. It should also be aligned with Edlund's customer relations. Concretely, MAL needed to support Edlund's service of providing its customers with template implementation of management actions. Therefore, a module system was implemented that made it easier for customers to pick and choose between standard management actions distributed across multiple files. This was done even though one test participant explicitly stated that it was easy to navigate in a MAL program since it was contained in a single file. Although this was arguably a paternalistic choice, we firmly believe that it is to the benefit of the users.

Lesson 5: The design resulting from phase one will likely contain obvious flaws easily discovered when testing it with an external domain expert.

Lesson 6: Implementing any language functionality will increase the cost of design revisions and make the design less flexible.

Lesson 7: Consider how to teach the language when designing the language. The teaching of the language is almost as important as the language itself and easily forgotten.

C. Domain evolution

The domain of MAL evolved during both design phases causing changes to the language design and its implementation. There were several causes to this evolution. One cause was regular software maintenance and refactoring, leading to small functionality changes. Another cause was novel domain discoveries leading to new data models and functionality, e.g. it turned out that it should be possible to model a policy by a probabilistic three-state entity. Finally, some evolution was caused by users' wishes. They wanted a clearer understanding of the projection platform, improved debugging facilities, and more control over the projection. At the beginning of the project, this evolution was straightforward to handle since we had no implementation. When implementation began in the second phase, evolution came at a high cost. This cost included development time on improving functionality and time spent updating existing MAL programs when breaking updates were made. It is possible that focusing more on the tooling of the technical design could have reduced some of this cost, e.g., by using a projectional editor. Nonetheless, these experiences reinforce our belief that one should try to delay implementation until it is necessary for some objective.

D. Experienced problems

If the only purpose of our design process was to ensure the usability of MAL, then we find it adequate. However, it is also a success criterion for an ADSL project that the language is used and actually alleviate the pains it is intended to. Although we are currently integrating MAL into Edlund's projection platform, we find it necessary to discuss our experienced problems with taking the language into production.

One problem with the design method, and implementation, is that while MAL was developed, actuaries had invested a significant amount of time into solutions written in a GPL. Although we believe MAL demonstrates significant improvements, actuaries may judge that these improvements do not offset the time invested into their current solutions [37]. Therefore, it would have been desirable to either start the development of MAL at an earlier point in time or to speed up the development.

Another solution to this problem could have been to have a more participatory design approach by involving Edlund's customers more directly in the design process [61]. However, such inclusion was not possible for us since it could potentially strain customer relations. Therefore, the future of the language depends on whether it demonstrates benefits significant enough that it can be introduced without fear of straining customer relations.

VIII. Threats to validity

The experiences described in the case study are the empirical findings presented in this paper. As explained in Section II, these findings should not be seen as an evaluation of the proposed method but as the material used to synthesise the method. This raises the possible external threat to validity that our experiences are not generalisable and the internal threat to validity that we are biased in the case study. We deal with each threat separately.

First, it is possible that our case study, and therefore the proposed method, is not generally applicable. This risk stems from the possibility of simply tailoring a design methodology to the specific design situation. To mitigate this risk, we have sought to be as precise as possible in describing the design context of MAL and identifying general challenges to the design process which are applicable to other ADSLs. At the same time, we have described our in-depth context-dependent experiences through a case study motivating our

methodological choices. Ultimately, future evaluation with other projects is needed to get a fuller understanding of the method.

Second, it is possible that we as authors have an inherent confirmation bias in presenting our case study: we believe that the design method is effective and leads to good ADSLs, and may selectively present only supporting evidence. But in fact, we have sought to describe a method that can mitigate specific problems and have openly discussed our experiences and problems using the method, with two goals: First, we hope that some practitioners may learn from our experiences. Second, we hope that this article will provoke other practitioners to more explicitly discuss their design processes for domain-specific languages and challenge ours.

IX. Conclusion

In this paper, we have described our experiences with conducting human-centred design to create an ADSL for non-programming experts in an evolving domain. We have in two ways described the characteristics of ADSLs, which we have found important for the design process. First, we have derived a language classification of programming languages based on their evolutionary characteristics. We argue that ADSLs belong to the class of steadily evolving *E*-type languages highly influenced by their application domain. Second, we have identified challenges to the design process. Based on these, we have argued that a design method must be able to handle evolving domains as well as include user validation in the design process while minimising user participation when possible.

We have conducted a case study on the design process of MAL and how this process sought to realise MAL's expected benefits. We found that early usage of rapid prototyping using pseudocode prototypes allowed us to explore a large design space. Using an in-team domain expert to conduct low-validity evaluations of these language designs allowed us to identify and fix usability issues. The low cost of the prototypes had the additional benefit of allowing rapid evolution of MAL's domain. Later in the process, domain experts external to the team was used for a higher validity evaluation of the language. These explorations indicate that users enjoy how they can model data, the conciseness of expressions, and find the language adequate in its expressiveness and functionality. Even more importantly, the evaluations pointed us to concrete usability issues, which the in-team expert did not uncover. However, we did experience problems in the design process. Once we began implementing the language, it became more expensive to handle domain evolution. Also, we experienced obstacles in taking the language into production primarily due to the perceived high costs of transitioning to MAL for Edlund's customers.

Based on these experiences, we have proposed a two-phase design method that seeks to guide ADSL designers in their innumerable syntactic and semantic design choices. An initial low-validity exploration using pseudocode prototypes allows the in-team expert to remain non-committal on design questions for as long as possible. A following higher-validity exploration seeks to ensure that the language design is generally usable by domain experts. Conclusively, the method seeks to incorporate the user's perspective into the design process while minimising the cost of conducting evaluations, thereby avoiding unnecessary overhead. We are currently looking into the possibility of conducting short co-design workshops with domain experts to design quality assurance measures for ADSLs. Future work will seek to evaluate this method by applying it to the design of other ADSLs.

References

[1] M. Voelter, *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. Lexington, KY: CreateSpace Independent Publishing Platform, Jan. 2013.

[2] "ACM Workshop on Real World Domain Specific Languages 2019," May 2021, accessed on: May 13, 2019. [Online]. Available: https://sites.google.com/site/realworlddsl

[3] B. Flyvbjerg, "Five Misunderstandings About Case-Study Research," *Qualitative Inquiry*, vol. 12, no. 2, pp. 219–245, Apr. 2006, publisher: SAGE Publications Inc.

[4] F. P. Brooks, "Keynote address: language design as design," in *History of programming languages—II*. New York, NY, USA: Association for Computing Machinery, Jan. 1996, pp. 4–16.

[5] E. Dijkstra, "Programming considered as a human activity," in *Classics in software engineering*. USA: Yourdon Press, Jan. 1979, pp. 1–9.

[6] C. A. R. Hoare, "Hints on programming language design." Stanford University, Stanford, CA, USA, Technical Report, 1973.

[7] B. A. Myers, J. F. Pane, and A. Ko, "Natural programming languages and environments," *Communications of the ACM*, vol. 47, no. 9, pp. 47–52, Sep. 2004.

[8] J. F. Pane and B. A. Myers, "Usability Issues in the Design of Novice Programming Systems," School of Computer Science, Carnegie-Mellon University, Pittsburg, Pennsylvania, Tech. Rep., Aug. 1996.

[9] L. McIver and D. Conway, "Seven Deadly Sins of Introductory Programming Language Design," in *Proceedings of the 1996 International Conference on Software Engineering: Education and Practice (SE:EP '96)*, ser. SEEP '96. USA: IEEE Computer Society, Jan. 1996, p. 309.

[10] D. Moody, "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 756–779, Nov. 2009, conference Name: IEEE Transactions on Software Engineering.

[11] D. van der Linden and I. Hadar, "A Systematic Literature Review of Applications of the Physics of Notations," *IEEE Transactions on Software Engineering*, vol. 45, no. 8, pp. 736–759, Aug. 2019, conference Name: IEEE Transactions on Software Engineering.

[12] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon, "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools," *Computer*, vol. 49, no. 7, pp. 44–52, Jul. 2016.

[13] D. Norman, *The Design of Everyday Things: Revised and Expanded Edition*, revised edition ed. New York, New York: Basic Books, Nov. 2013.

[14] A. Stefik, B. Sharif, B. A. Myers, and S. Hanenberg, "Evidence About Programmers for Programming Language Design (Dagstuhl Seminar 18061)," *Dagstuhl Reports*, vol. 8, no. 2, pp. 1–25, 2018, place: Dagstuhl, Germany Publisher: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[15] M. Coblenz, J. Aldrich, B. A. Myers, and J. Sunshine, "Interdisciplinary programming language design," in *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2018. Boston, MA, USA: Association for Computing Machinery, Oct. 2018, pp. 133–146.

[16] A. Stefik and S. Hanenberg, "Methodological Irregularities in Programming-Language Research," *Computer*, vol. 50, no. 8, pp. 60–63, 2017, conference Name: Computer.

[17] I. Poltronieri Rodrigues, M. de Borba Campos, and A. F. Zorzo, "Usability Evaluation of Domain-Specific Languages: A Systematic Literature Review," in *Human-Computer Interaction. User Interface Design, Development and Multimodality*, ser. Lecture Notes in Computer Science, M. Kurosu, Ed. Cham: Springer International Publishing, 2017, pp. 522–534.

[18] S. Hanenberg, "Empirical, Human-Centered Evaluation of Programming and Programming Language Constructs: Controlled Experiments," in *Grand Timely Topics in Software Engineering*, ser. Lecture Notes in Computer Science, J. Cunha, J. P. Fernandes, R. Lämmel, J. Saraiva, and V. Zaytsev, Eds. Cham: Springer International Publishing, 2017, pp. 45–72.

[19] S. Kleinschmager, R. Robbes, A. Stefik, S. Hanenberg, and E. Tanter, "Do static type systems improve the maintainability of software systems? An empirical study," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, Jun. 2012, pp. 153–162.

[20] A. Stefik and S. Siebert, "An Empirical Investigation into Programming Language Syntax," *ACM Transactions on Computing Education*, vol. 13, no. 4, pp. 19:1–19:40, Nov. 2013.

[21] A.-J. Kaijanaho, "Evidence-based programming language design : a philosophical and methodological exploration," *Jyväskylä studies in computing*, no. 222, 2015, accepted: 2015-11-17T10:33:20Z ISBN: 9789513963880 Publisher: University of Jyväskylä.

[22] A. Blackwell, M. Burnett, and S. Jones, "Champagne Prototyping: A Research Technique for Early Evaluation of Complex End-User Programming Systems," in *2004 IEEE Symposium on Visual Languages - Human Centric Computing*. Rome: IEEE, 2004, pp. 47–54.

[23] J. D. Gould, J. Conti, and T. Hovanyecz, "Composing Letters with a Simulated Listening Typewriter," *Proceedings of the Human Factors Society Annual Meeting*, vol. 25, no. 1, pp. 505–508, Oct. 1981, publisher: SAGE Publications.

[24] T. Marter, P. Babucke, P. Lembken, and S. Hanenberg, "Lightweight programming experiments without programmers and programs: an example study on the effect of similarity and number of object identifiers on the readability of source code using natural texts," in *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2016. Amsterdam, Netherlands: Association for Computing Machinery, Oct. 2016, pp. 1–14.

[25] M. Fowler, *Domain Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.

[26] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, Dec. 2005.

[27] R. Lämmel, *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer International Publishing, 2018.

[28] A. Barišić, V. Amaral, M. Goulao, and A. Aguiar, "Introducing usability concerns early in the DSL development cycle: FlowSL experience report," p. 10.

[29] A. Barišić, V. Amaral, M. Goulão, and B. Barroca, "Quality in use of domain-specific languages: a case study," in *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, ser. PLATEAU '11. Portland, Oregon, USA: Association for Computing Machinery, Oct. 2011, pp. 65–72.

[30] V. Zaytsev, "Language Design with Intent," in *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Sep. 2017, pp. 45–52.

[31] J. L. C. Izquierdo and J. Cabot, "Community-driven language development," in *2012 4th International Workshop on Modeling in Software Engineering (MISE)*, Jun. 2012, pp. 29–35, iSSN: 2156-7891.

[32] M. J. Villanueva, F. Valverde, and O. Pastor, "Involving End-Users in the Design of a Domain-Specific Language for the Genetic Domain," in *Information System Development*, M. José Escalona, G. Aragón, H. Linger, M. Lang, C. Barry, and C. Schneider, Eds. Cham: Springer International Publishing, 2014, pp. 99–110.

[33] M. Coblenz, G. Kambhatla, P. Koronkevich, J. L. Wise, C. Barnaby, J. Sunshine, J. Aldrich, and B. A. Myers, "PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design," *arXiv:1912.04719 [cs]*, Aug. 2020.

[34] T. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *J. Vis. Lang. Comput.*, 1996.

[35] S. Clarke, "Evaluating a new programming language," *13th Workshop of the Psychology of Programming Interest Group*, pp. 275–289, 2001.

[36] S. P. Jones, A. Blackwell, and M. Burnett, "A user-centred approach to functions in Excel," in *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ser. ICFP '03. Uppsala, Sweden: Association for Computing Machinery, Aug. 2003, pp. 165–176.

[37] A. Blackwell and M. Burnett, "Applying attention investment to end-user programming," in *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, Sep. 2002, pp. 28–30.

[38] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning, "The State of the Art in Language Workbenches," in *Software Language Engineering*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, M. Erwig, R. F. Paige, and E. Van Wyk, Eds. Cham: Springer International Publishing, 2013, vol. 8225, pp. 197–217, series Title: Lecture Notes in Computer Science.

[39] M. Voelter, B. Kolb, T. Szabó, R. Daniel, and A. van Deursen, "Lessons learned from developing mbeddr: a case study in language engineering with MPS," *Software & Systems Modeling*, 2017.

[40] D. Ratiu, M. Voelter, and D. Pavletic, "Automated testing of DSL implementations—experiences from building mbeddr," *Software Quality Journal*, vol. 26, no. 4, pp. 1483–1518, Dec. 2018.

[41] A. M. Şutîi, M. v. d. Brand, and T. Verhoeff, "Exploration of modularity and reusability of domain-specific languages: an expression DSL in MetaMod," *Computer Languages, Systems & Structures*, vol. 51, pp. 48–70, Jan. 2018.

[42] N. Vasudevan and L. Tratt, "Comparative Study of DSL Tools," *Electronic Notes in Theoretical Computer Science*, vol. 264, no. 5, pp. 103–121, Jul. 2011.

[43] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, Sep. 1980, conference Name: Proceedings of the IEEE.

[44] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, and P. Naur, "Report on the algorithmic language ALGOL 60," *Communications of the ACM*, vol. 3, no. 5, pp. 299–314, May 1960.

[45] N. Wirth and C. A. R. Hoare, "A contribution to the development of ALGOL," *Communications of the ACM*, vol. 9, no. 6, pp. 413–432, Jun. 1966.

[46] N. Wirth, "The programming language pascal," *Acta Informatica*, vol. 1, no. 1, pp. 35–63, Mar. 1971.

[47] "Perl Compatible Regular Expressions," accessed on: May 13, 2019. [Online]. Available: http://www.pcre.org/

[48] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[49] D. May, "Occam," Apr. 1983.

[50] J. Armstrong, "The development of Erlang," in *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ser. ICFP '97. Amsterdam, The Netherlands: Association for Computing Machinery, Aug. 1997, pp. 196–203.

[51] "Tyche modelling platform," accessed on: May 13, 2019. [Online]. Available: https://www.rpc-tyche.com/Software/Modelling

[52] S. Peyton Jones, J.-M. Eber, and J. Seward, "Composing contracts: an adventure in financial engineering (functional pearl)," *ACM SIGPLAN Notices*, vol. 35, no. 9, pp. 280–292, Sep. 2000.

[53] J. Andersen, E. Elsborg, F. Henglein, J. G. Simonsen, and C. Stefansen, "Compositional specification of commercial contracts," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 6, pp. 485–516, Oct. 2006.

[54] D. A. Ladd and J. C. Ramming, "Two Application Languages in Software Production," p. 9.

[55] I. Sommerville, *Software Engineering*, 9th ed. USA: Addison-Wesley Publishing Company, 2010.

[56] Evans, *Domain-Driven Design: Tacking Complexity In the Heart of Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[57] T. K. Landauer and J. Nielsen, "A Mathematical Model of the Finding of Usability Problems," *INTERCHI*, p. 8, 1993.
[58] K. Bruhn and A. S. Lollike, "Retrospective reserves and bonus," *Scandinavian Actuarial Journal*, pp. 1–19, Aug. 2020.
[59] D. K. Falden and A. K. Nyegaard, "Retrospective Reserves and Bonus with Policyholder Behavior," *Risks*, vol. 9, no. 1, p. 15, Jan. 2021, number: 1 Publisher: Multidisciplinary Digital Publishing Institute.
[60] B. M. Jensen, M. D. Raffnsøe, and J. She, "Forsikrings- og pensionssektoren i ny kvartalsvis statistik," 2019.
[61] K. Bodker, F. Kensing, and J. Simonsen, *Participatory It Design: Designing for Business and Workplace Realities*. Cambridge, MA, USA: MIT Press, 2004.

Appendix B

Co-designing DSL Quality Assurance Measures for and with Non-programming Experts

Co-designing DSL Quality Assurance Measures for and with Non-programming Experts

Holger Stadel Borum
hstb@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

Christoph Seidl
chse@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

Peter Sestoft
sestoft@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

Abstract

Domain-specific languages seek to provide domain guarantees that eliminate many errors allowed by general-purpose languages. Still, a domain-specific language requires additional quality assurance measures to ensure that specifications behave as intended by the users. However, some domains may have specific quality assurance measures (e.g., proofs, experiments, or case studies) with little tradition of using quality assurance measures customary to software engineering. We investigate the possibility of accommodating such domains by conducting a workshop with 11 prospective users of a domain-specific language named MAL for the pension industry. The workshop emphasised the need for supporting actuaries with new analytical tools for quality assurance and resulted in three designs: *quantity monitors* let users identify outlier behaviour, *fragment debugging* lets users debug with limited evaluative power, and *debugging spreadsheets* let users visualise, analyse, and remodel concrete calculations with an established domain tool. Based on our experiences, we hypothesise that co-design workshops are a viable approach for DSLs in a similar situation.

CCS Concepts: • **Software and its engineering → Domain specific languages**.

Keywords: domain-specific language, co-design

ACM Reference Format:
Holger Stadel Borum, Christoph Seidl, and Peter Sestoft. 2021. Co-designing DSL Quality Assurance Measures for and with Non-programming Experts. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Domain-Specific Modeling (DSM '21), October 18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3486603.3486776

DSM '21, October 18, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9106-1/21/10…$15.00
https://doi.org/10.1145/3486603.3486776

1 Introduction

Quality assurance is an important software engineering practice that ensures that developed software behaves as expected in different contexts. While domain-specific languages (DSLs) and models seek to eliminate many erroneous behaviours permissible by general-purpose languages (GPLs), they do not eliminate the need for quality assurance. Because user errors made in a DSL may have serious consequences (e.g., when guiding financial decisions), the design of quality assurance measures should be an integral part of DSL design. When users do not have a background in software engineering, it is not apparent which measures they deem viable. Users may come from fields with quality assurance measures that do not align directly with traditional software engineering practices, e.g., proofs, experiments, or case studies. This possible discrepancy creates a need for actively involving users in the design of quality assurance measures to ensure that a) the measures support users in their quality assurance and b) users will find the designed quality assurance measures valuable and use them.

In this paper, we first discuss and investigate state-of-the-art for co-designing DSLs (Section 2). Then we describe the DSL named Management Action Language or MAL (Section 3), for which we want to design quality assurance measures. The purpose of MAL is to provide customers of Edlund A/S with a user-friendly and efficient way of specifying so-called management actions in the projection of the asset/liability balance of a pension company. These projections are a form of risk management that ensures a pension company remains solvent when using a management action strategy. Therefore, it is of great importance that a MAL program accurately models real management actions, such as how to handle varying yields of investments on pension products with an interest guarantee. While MAL's prospective users have a strong mathematical background, some have limited programming experience, which means they come from a field where proofs, peer discussions, and problem analysis are important quality assurance measures. To accommodate this background, we held a co-design workshop with 11 prospective users on the quality assurance of asset/liability projections (Section 4). Since all workshop participants come from customer companies of Edlund, the workshop was not merely an experiment in co-design but an actual step in

the development and deployment of MAL with the corresponding risk of straining Edlund's customer relations. The workshop sought to engage users actively in designing quality assurance measures to ensure the measures matched the customers' workflow. The workshop, combined with our prior domain knowledge, resulted in the design of three quality assurance measures. First, quantity monitors can be used to identify unexpected or outlier behaviour for users to examine (Section 5). Second, fragment debugging allows users to debug MAL code without having direct access to the entire asset/liability projection (Section 6). Third, interactive debugging spreadsheets lets users visualise and analyse program behaviour within a comfortable setting of spreadsheets (Section 7). We conclude that the co-design workshop was a viable approach for creating quality assurance measures for MAL leading to designs tailored to the domain, and we hypothesise that the approach is viable to overcome similar challenges for other DSLs (Section 8).

The contributions of this paper are both an empirical co-design workshop experience and the connected constructive designs derived from the workshop. Concretely, our contributions are:

- A presentation of MAL to be used in asset/liability projections
- An approach to and experiences with co-designing quality assurance measures through a user workshop demonstrating how three concrete quality measures were derived from the co-design workshop.
- Debugging spreadsheets as a general quality assurance measure applicable to domains with complex mathematical calculations.

2 State of the Art

We understand *co-design* (or collaborative design) as a design methodology where designers, implementers, and users collaborate to create a design [9]. Co-design may be more or less pervasive in a design process in terms of who collaborates, when they collaborate, and to what degree non-designers collaborate on even footings with the designer [9]. Our work may be considered only as a one-off workshop where users participate equally powerful as the designer, but we do consider a single co-design workshop as being significantly more collaborative than having none. In that sense, this work focuses on collaborative design generation, and it may be more suitable to label our work as *co-creation* [26]. Although co-design is mentioned partially as a rebranding of *participatory design* [6] [26], we refrain from using this term since it suggests a more holistic design methodology where stakeholder analysis, vision anchoring, and vision alignment play an essential role.

It is difficult to find papers that explicitly deal with co-designing DSLs. In fact, we have found only a single case discussing co-designing a DSL for community-supported appliances [24]. Unfortunately, this work focuses primarily on the resulting design and very little on the co-design process leading to the language.

However, there exists work on language co-design if we broaden our view to include work that does not itself claim to conduct co-design. *Example-driven meta-model development* seeks to include domain experts generatively in the design process by inducing meta-models from examples created by the domain experts [29] [18] [22]. In this method, the software engineer takes on a facilitating design role using their meta-modelling expertise to monitor and guide the design process. With other similar methods, a meta-model and potentially a modelling tool is created from free-form modelling [21] [10]. *Natural programming* is a similar bottom-up method for designing programming languages [25]. Here a programming language design is created based on pseudocode solutions written by users. This method is similar to the advice of *looking at existing notations* but where we would hardly consider following this advice as conducting co-design. Non-software product lines can be seen as a form of co-creation late in a product development process [26], and similarly, software product lines [4] for DSLs [32] can be seen as a form of late-stage DSL co-creation. However, we think that earlier user collaboration in the design process is necessary for us to meaningfully talk about co-design. The DSL named *Collaboro* [17] was created to involve communities in the design process of DSLs. Other work investigates how to involve users non-generatively in the DSL design process [33] or evaluates modelling tools empirically [30]. We have done similar work with MAL [8].

With regards to quantity monitors and fragment debuggers, the main contribution of this paper is the process of deriving these as suitable quality assurance measures, not the innovation of the concepts themselves. Quantity monitors have been used as a quality assurance measure in many DSLs and seem especially popular within the robotics community [14] [3]. We speculate that this popularity is because the domain of robotics shares the characteristic with asset/liability projections of being challenging to define correct behaviour in. For fragment debuggers, more advanced debuggers for distributed asynchronous systems had already been proposed and developed by the '90s [28]. Also, *time-travelling* debuggers have extensively used a record/replay idea to allow users to step back execution time [5].

To our best knowledge, the concept of using spreadsheets to debug programs is novel. Although there exist many tools helping end-users to debug and test spreadsheets [13] [1] [2], this functionality is somewhat the opposite of what we are

proposing. Other work explores different ways of visualising programs [31] and traces [23] [12], but we have found no examples of using spreadsheets to do so. During our examination of recent US patents concerning spreadsheet implementation [7], we found no such functionality.

3 Management Action Language

Edlund is a company that creates software solutions for pension companies. MAL is designed as a part of the *solvency projection platform* that actuaries use as a risk-management tool for customer companies. A single projection consists of two subprocedures: First, a *projection step* is performed by the *projection engine* that transforms quantities of a company one timestep into the future. Second, a *management step* is performed that mimics the management made by the pension company. In the management step, a company can, for example, choose how to distribute investment yields to policyholders. These two subprocedures are iteratively executed until the projection's endpoint. Since a single projection is parametrised on an economic scenario, the projection platform essentially performs a Monte Carlo simulation using different economic scenarios.

While pension companies can use the same *projection step*, they each have their own *management step* that models the business rules of the company. Currently, these business rules are written in a general-purpose language. The primary purpose of MAL (see Figure 1) is to afford actuaries with an easier way to model business rules in a manner that allows them to be executed efficiently by the projection platform. This affordance primarily comes from the following:

The inheritance-based data declarations let users model data in an object-oriented fashion and use union types as a fine-grained mechanism to group similar extensions. E.g., if a user creates the two cash flow extensions `Foo` and `Bar`, then they may read from or update fields shared by these on the union type `{Foo|Bar}`.

The expression language lets users declaratively describe the actuarial mathematics of a projection. We observed basic arithmetic, function application, and mappings as the primary vocabulary used by actuaries discussing management actions.

The module system lets users split their computations into units and reuse these in different projections. Furthermore, the module system allows Edlund to maintain a standard library of template actions that customers may modify. An example of such a template action could be how to calculate the solvency capital requirement of a company.

Code generation cleanly decouples application logic from business logic. This decoupling hides messy details of general-purpose solutions such as interfaces from users. Simultaneously, it allows Edlund to make some changes to the underlying software platform without users noticing.

4 Co-design Workshop

In 2019, the Danish pension industry managed assets for 200% of Denmark's BNP [19] making it important for prospective users of MAL to accurately model business rules. Our work with designing MAL left us with a many-faceted picture of its prospective users. Briefly, prospective users have a strong mathematical background and use mathematical models originating from ongoing actuarial research with several unanswered questions. Users regularly use spreadsheets as part of their analytical and experimental work. Many users have limited software engineering experience and correspondingly limited experience with software testing practices such as unit, regression, or property-based testing. Still, users want to account for tiny fractions of Danish Kroner and ensure that calculations pass so-called Martingale tests. This picture, combined with our non-expert domain understanding, made it difficult for us to design quality assurance measures. Although we could hope to adopt test practices from software engineering in MAL, we had concerns about whether such facilities would be suitable for the domain and, even more importantly, whether users would appreciate them. To mitigate this risk, we decided to conduct a co-design workshop to include users directly in the design of quality assurance measures.

4.1 Plan

We invited actuaries from customer companies to participate in a workshop on quality assurance of asset/liability projections. The invitation purposefully did not mention whether we targeted the current general-purpose or future domain-specific solution to avoid participants getting hung up on this difference. Our intention was to focus on designing quality assurance measures in general and such measures could apply to both settings with different implementation strategies. We prepared three activities progressing from the descriptive to the normative. The movement from eliciting *what is* to investigating *what can be* would allow participants to engage with different levels of creativity. First, we would ask participants to sketch and present their current approach to quality assurance. Second, we would ask participants to identify kinds of properties to ensure and specific properties of projections. During this activity, we had different kinds of properties prepared to facilitate the discussion. Finally, we would ask participants for approaches to ensure these properties. During this activity, we were prepared to sketch different traditional approaches to testing to facilitate the

t ::= Tag	
τ ::= Type	
x ::= Identifier	
v ::= n	Integer
| f	Float
| s	String
e ::= v | x	
| $f(e, \ldots)$	Function application
| $if(e, e, e)$	Conditional
| $e.x$	Projection
| $e : \{t, \ldots\}$	Tag filter
| `map` x `in` e `with` { e }	Map
| `match` x `with` | $t\ x$ -> e | ...	Tag match
| $e@x$	Map + Projection
s ::= e <| e <| e	Reserve transfer
| `update` x `in` e `with{` s `}`	Update iteration
| `let` x = e	Let binding
| $e.x$ = e	Assignment
| `do` $x(e, \ldots)$	Procedure call
| s ...	Block
o ::= `CashFlow` | `Reserve`	
d ::= `action` $x(x : \tau, \ldots)$`with{`s`}`	Action
| `fun` $x(x : \tau, \ldots) = e$	Function
| `data` t < `extends` t > {$x : \tau$ <, output as o >, ...}	Data
| `import` x	Import
m ::= `module` x d ...	Module
| `main` x d ...	Main module

Figure 1. A subset of MAL's grammar containing the most important language constructs. **Legend:** '...' means repeated productions. '<' and '>' delimit an optional production.

discussion. Participants were not asked to prepare anything in advance and were asked to use whiteboard drawings as a means of communication to welcome off-the-top-of-the-head ideas and discussions.

4.2 Execution

Eleven people working with asset/liability projections participated in the workshop, which was held virtually due to COVID-19 restrictions. An online whiteboard application was used as the interactive medium. Unfortunately, multiple people faced technical issues using the whiteboard (e.g., firewall setups and cross-organisational access permissions). These restrictions made conversation and activities less fluid, but, luckily, participants were willing to put in the effort to overcome these challenges. We refrained from recording the session as to not impede participants' willingness to participate in the open discussion. Therefore, the quotations in the following text are not ad verbum but as close as possible.

Existing Quality Assurance Measures. All companies relied on external calculations (often performed in a spreadsheet) to check that implemented management actions behaved as intended. A common approach was to start with an elementary external scenario which was incrementally made more advanced and realistic by incorporating more and more advanced data and management actions. This approach was reported to be well-suited for identifying errors such as forgetting to implement parts of a calculation or missing a negation. At the time, companies made limited use of unit and regression tests, but this usage could be increased over time. It was reported that errors were rarely discovered using these kinds of tests. Finally, many errors occurred in the interface with the projection engine, and these errors were difficult to debug. During the discussion, some participants were hesitant to embrace automatic testing, as one said: "I am afraid of relying only on automatic tests because manual tests provide new insights [to the understanding of management actions]". This difference between us thinking of quality assurance in terms of software passing a good test suite and participants thinking of it as ensuring a deep understanding of management actions was pervasive for the entire workshop.

From Testing to Analyses. Participants struggled when asked to try to identify types of properties or projection-specific properties, such as the total reserve must equal the sum of all discounted future cash flows. Even when concretely asked if there were any guarantees to be made between two versions of external spreadsheet calculations, participants could not find any. One participant said: "I would love to list different properties, but the calculations are so complex that I am simply unable to do so". This development was, put mildly, problematic for the remaining workshop that assumed we could at least identify some properties or property types. After some thought, we chose to shift focus from identifying properties to the more general question of "how do you think we can improve existing quality assurance?" Although this question was not originally planned, it progressed the workshop and led to a thematic shift in the workshop, moving from various testing approaches to analytical tools.

Participants all seemed to agree that they could use better tools to understand management actions. They did not need

improved support for testing but needed more support to understand specified models and calculations. Three concrete qualitative measures appeared as a result of this discussion. First, one participant wanted to identify and examine outlier behaviour by "for example, looking for values that diverge from the [Monte Carlo] average by, say, more than three standard deviations". Such behaviour could be benign but interesting to examine more closely, especially since such outliers could significantly impact the average. This discussion led to the design of *quantity monitors* (Section 5). Second, participants sought improved facilities for live debugging since their current setup is hindered by limited access to the projection engine. This wish led to further work with *fragment debugging* (Section 6). Third, based on the discussion, we proposed that it could be possible to export calculations from a DSL or GPL program to a spreadsheet for further investigation. Participants showed interest in such functionality, even when discussed as a relatively vague concept. These discussions led to the design of *debugging spreadsheets* (Section 7).

5 Quantity Monitor

From the workshop, we learned that while domain experts have an in-depth understanding of their domain, they find it difficult to state precise properties about their management actions when prompted. This absence of precise, interim properties makes it difficult to test solutions and impossible to perform conventional property-based testing [11]. However, domain experts still have an intuition of how their domain behaves, which they want to use to monitor the execution of a program. A *quantity monitor* lets domain experts express this intuition as Boolean predicates that can identify scenarios where the domain behaves counter-intuitively. Such behaviour may either be caused by a modelling error or a benign misunderstanding of the domain. In both cases, the behaviour warrants further examination. For quantities approximated using a Monte Carlo simulation, it is possible to leverage the simulation to look for outlier behaviour in concrete Monte Carlo runs. By assuming that an observation close to the observed average is either correct or benign, we may look for outlier observations far from the average to examine. We refer to such observations as *crosscutting* since they crosscut simulations.

5.1 Specification and Report

In MAL, a quantity monitor could be specified with a loop-like notation, as seen in Figure 2. The `monitor`-construct consists of a list of Boolean expressions that specifies the monitored properties. The example in Figure 2 states that 1) the reserve of a policy remains non-negative, 2) a policy always belong to exactly one interest group, and 3) the reserve of a policy does not exceed five standard deviations above the Monte Carlo average of the policy's reserve. While the first two properties are reminiscent of classical assert statements and could be implemented as such, the third property introduces many complications since it requires property checking across multiple Monte Carlo simulations. We intentionally designed MAL to encapsulate a single Monte Carlo simulation and thereby disallowing one simulation from depending on others. However, if users are allowed to monitor only single runs in isolation and the aggregated result, it is possible that errors may hide in the aggregation. Therefore, users are allowed to specify crosscutting properties with the sampling consequences discussed in Section 5.2.

```
monitor p in Policies
{
  0 <= p.Reserve
  count(p.Groups:Interest) = 1
  p.Reserve < MC.avg(p.Reserve)
            + 5 * MC.sd(p.Reserve)
}
```

Figure 2. A policy monitor specified in MAL.

Quantity monitors may be used to generate a monitor report that lists instances where properties do not hold during a projection. A domain expert may both use a monitor report to identify scenarios that need to be examined and as a testament to the quality of their management actions. For this latter purpose, a domain expert may find it acceptable that a property does not always hold and find it valuable to document how often the property holds.

5.2 Monitoring Strategies

A quantity monitor comes with a trade-off between its precision and its performance cost. The cost of monitoring is especially significant for crosscutting properties. For these properties, a substantial amount of data has to be stored during execution to compare the individual value with its aggregate. Managing this kind of data is especially cumbersome for more expensive simulations performed in a distributed setup. Here we describe four monitoring strategies with their respective pros and cons.

Total monitoring checks all updates made to monitored quantities. This strategy guarantees to discover if a property does not hold at some time during a specific projection. However, the strategy is costly since it requires a lot of additional program evaluation and stored data for simulation cutting properties.

Result monitoring checks that properties hold at the beginning and at the end of a projection. This strategy provides no guarantees during execution and could almost be implemented as pre and post-processing by the users themselves.

However, the strategy is computationally cheap since it almost requires no extra data nor evaluation.

Random, heuristic, and explicit monitoring all seek to strike a balance between the guarantees provided by the monitor and the cost of doing so. *Random monitoring* samples at random points during execution. *Heuristic monitoring* samples in accordance to some metric, such as at least 50% percentage of values have changed since the last sample. *Explicit monitoring* lets users define when to monitor with explicit monitor statements.

6 Fragment Debugging

After a quantity monitor has identified suspicious behaviour to investigate, the user needs tools for analysing the behaviour. At the workshop, we discussed classical live debugging and debugging spreadsheets (Section 7) as quality assurance means to inspect and analyse worrisome behaviour. While live, step-by-step debugging functionality is available in most development environments, such functionality may be limited for a DSL that expresses only program fragments. Users may have evaluative powers to execute only DSL fragments, with the remaining execution being unavailable due to IP protection, cost of maintenance, security concerns, or other worries regarding a customer relationship. This means that execution may either take place *locally* on a users' machine or *remotely* on a server, possibly in the cloud. To remain general, we say that some execution may be performed by an execution engine that corresponds to the projection engine for MAL. We identify five different approaches to step-by-step fragment debugging:

Local debugging and remote debugging correspond to a traditional debugging where all evaluation is executed by a single machine (a,b in Table 1). For our purposes, the local setup is uninteresting since it requires users to have full evaluative powers. In contrast, the full remote setup is feasible but requires that the remote setup is implemented with such functionality in mind as it requires the setup to communicate following a specified debug protocol.

Live distributed debugging has an execution split between the execution engine and the fragment debugger (c in Table 1). With this approach, the execution engine calls the debugger whenever it requires a DSL fragment to be executed. This approach allows users to make live code and value changes during debug execution. However, an execution engine may not have been implemented with this functionality in mind, and it may therefore not be able to defer execution to a remote environment when required.

Prerecorded debugging starts with a normal remote program execution where the execution engine is responsible for evaluating DSL fragments (d in Table 1). Whenever a DSL fragment is evaluated, the execution engine records the state relevant for this evaluation. After execution, these recorded states may be used by the user's debugger to simulate the execution engine. In this simulation, the DSL fragments may be reevaluated, allowing the user to experiment with changing values. However, these changes will not affect the prerecorded execution. There are two other downsides to this approach. First, the engine must be able to record relevant states, and the additional data may slow down execution. Second, the user will have to wait for an entire program execution before being performing any debugging.

Table 1. Approaches to DSL fragment debugging showing where fragments (F) and execution engine (E) is executed.

	a	b	c	d,e
Local	F,E		F	F
Remote		F,E	E	F,E

Fast forward debugging is essentially the same as *prerecorded debugging*, where a new recording is made to handle live code and value changes (e in Table 1). Although this approach provides users with greater flexibility, the flexibility comes with a performance cost. Also, the evaluation engine may have to be significantly altered to be able to either handle changes occurring midway during executions or starting midway execution. If the latter is possible, then it seems like it should be possible to support *live distributed debugging*.

7 Debugging Spreadsheets

One conclusion of our design workshop is that Danish actuaries profoundly and happily use spreadsheets for modelling, analysis, and calculations. Spreadsheet applications shine in their ability to visualise concrete calculations and interactively recalculating them. There are several features that seek to introduce abstractions to spreadsheets, such as sheet-defined functions [20] [27], anonymous functions [16], macros, and external scripts. However, these abstractions are most suitable to be used as part of concrete calculations and not as a mechanism to specify general programs. In this section, we will show how spreadsheets can be used to debug concrete MAL calculations. We call such a spreadsheet a *debugging spreadsheet*. We first show an example demonstrating how a debugging spreadsheet can be derived from an execution of a MAL program and then move on to presenting a debugging-spreadsheet semantics for MAL. Although the debugging-spreadsheet semantics is presented for MAL, it should be evident that a similar approach is possible for other languages and seems especially appropriate for functional and arithmetic heavy languages.

```
update policy in Policies
{
  let baseFactor = pow(1 + Global.Param.BaseFee, Projection.PeriodLength) - 1
  policy.Fee = baseFactor * policy.TotalReserve
}
```

Figure 3. A MAL snippet that calculates a fee of all policies.

Table 2. A formula view of a part of the corresponding debugging spreadsheet of the MAL snippet in Figure 3. The loop is unrolled such that the iteration for `Policy 1` starts in `A1` and the iteration for `Policy 2` starts in `E1`.

	A	B	C	D	E
1	`Policy 1`				`Policy 2`
2			`Global.Param.BaseFee`	`Projection.PeriodLength`	
3	`let baseFactor =`	`=POWER(1+C3,D3)-1`	`0.02`	`1.3`	...
4			`policy.TotalReserve`		
5	`policy.Fee =`	`=B3*C5`	`5234.23`		...

7.1 Example

Imagine a scenario where an actuary observes that there is an erroneous fee of some policy (see Figure 3). If the actuary does not immediately find an error in the specification, then they must observe all values used in the calculation to identify the problem. Table 2 shows a debugging spreadsheet of an execution of the example program that allows users to investigate the error with an established domain tool. Note that the values corresponding to `baseFactor` and `policy.Fee` are calculated by the spreadsheet, which means it is possible for the user to further analyse the calculations.

7.2 Design Goals

The derivation of a debugging spreadsheet from a MAL execution should maximise:

1. *Recognisability*, i.e., the degree to which users can recognise their original computations.
2. *Completeness*, i.e., the degree to which MAL programs can be translated to a spreadsheet.
3. *Consistency*, i.e., the degree to which a user edit in a debugging spreadsheet is equivalent to an edit in the corresponding MAL program. Conversely, an *inconsistent* edit does not have an equivalent MAL edit.

As we will see, these parameters are not independent, at least not from a practical point of implementation. The main challenge is that as the completeness of a solution increases, it becomes more difficult to ensure consistency and to find a recognisable layout.

The layout of a debugging spreadsheet is a good starting point for our discussion and a key concern for recognizability. We use the design concept of mapping by letting a MAL-program line roughly correspond to a spreadsheet row with calculations extending from left to right. As a consequence, we unroll loops horizontally, as seen in Table 2. Such unrolling introduces the possibility of inconsistencies by having a copy of a formula for each unrolled iteration. However, such possible formula inconsistencies are to be expected by seasoned spreadsheet users.

Composite and mutable data (objects) can be represented in three ways. First, all data objects can be placed on a separate sheet and referenced as needed. When an object is updated, a new data entry is made on the sheet with new references pointing to this updated entry. Second, the spreadsheet can be augmented with both composite and mutable values making it possible to accurately represent data objects. Third, it is possible to observe whenever a value is read and later updated. Therefore, values can be placed directly in the debugging spreadsheet the first time they are used and when they are subsequently updated. We use the third approach since we believe the first approach would lower recognisability, and the second requires a non-standard spreadsheet implementation and may be exotic to even seasoned spreadsheet users.

Function applications can be represented by either inlining the function body or mimicking the function application in the spreadsheet. The inlining strategy is possible since concrete executions always terminate. Although the inline strategy introduces the same kind of possible inconsistencies as loop unrolling, it makes it possible to debug the function in the spreadsheet. Alternatively, there are multiple ways to mimic a MAL function in the spreadsheet. First, some numeric functions such as `+` and `max` can use their spreadsheet counterpart. Second, some user functions can be recreated as a spreadsheet function, as an anonymous function, or in an external scripting language. Third, as a last resort, MAL

$$\mathsf{E}[\![\mathsf{E}]\!] : \mathsf{env} \to \mathsf{cells}[,]$$
$$\mathsf{E}[\![\mathsf{n}]\!](\Gamma) = [] +_v \mathsf{NumberCell\ n}$$

$$\mathsf{E}[\![\mathsf{x}]\!](\Gamma) = \begin{cases} [] +_v \mathsf{CellRef\ c} & \text{if } \Gamma(x) = \mathsf{c} \\ \mathsf{MAL(x)} & \text{if } x \notin dom(\Gamma) \end{cases}$$

$$\mathsf{E}[\![\mathsf{e}_1.\mathsf{x}]\!](\Gamma) = \mathsf{MAL}(\mathsf{e}_1.\mathsf{x})$$

$$\mathsf{E}[\![\mathsf{f}(\mathsf{e}_1, \ldots, \mathsf{e}_n)]\!](\Gamma) = \begin{cases} [] +_v \mathsf{f}'(\mathsf{e}'_1, \ldots, \mathsf{e}'_n) +_h c_1 +_h \ldots +_h c_n & \text{if } ss(\mathsf{f}) = \mathsf{f}' \\ MAL(\mathsf{f}(\mathsf{e}_1, \ldots, \mathsf{e}_n)) +_h c_1 +_h \ldots +_h c_n & \text{if } \mathsf{f} \notin dom(ss) \end{cases}$$

$$\text{where } c_1 = \mathsf{E}[\![\mathsf{e}_1]\!](\Gamma)$$
$$\vdots$$
$$c_n = \mathsf{E}[\![\mathsf{e}_n]\!](\Gamma)$$

$$\mathsf{S}[\![\mathsf{S}]\!] : \mathsf{env} \to \mathsf{env} * \mathsf{cells}[,]$$

$$\mathsf{S}[\![\mathsf{let\ x\ =\ e}]\!](\Gamma) =$$
$$\quad \text{let } cs = \mathsf{E}[\![\mathsf{e}]\!](\Gamma)$$
$$\quad \Gamma[\mathsf{x} \mapsto cs[1,0]], [] +_v \mathsf{TextCell\ "let\ x\ ="} +_h cs$$

$$\mathsf{S}[\![\mathsf{e}_1.\mathsf{x} = \mathsf{e}_2]\!](\Gamma) =$$
$$\text{let } cs = \mathsf{E}[\![\mathsf{e}_2]\!](\Gamma)$$
$$\Gamma, [] +_v \mathsf{TextCell\ "e_1.x\ ="} +_h cs$$

$$\mathsf{S}[\![\mathsf{s}_1 \ldots \mathsf{s}_n]\!](\Gamma) =$$
$$\text{let } \Gamma_1, cs_1 = \mathsf{S}[\![\mathsf{s}_1]\!](\Gamma)$$
$$\vdots$$
$$\text{let } \Gamma_n, cs_n = \mathsf{S}[\![\mathsf{s}_n]\!](\Gamma_{n-1})$$
$$\Gamma_n, cs_1 +_v \ldots +_v cs_n$$

$$\mathsf{S}[\![\mathsf{update\ x\ in\ e\ with\ s}_1\ \mathsf{end}]\!](\Gamma) =$$
$$\text{let } [\mathsf{v}_1, \ldots, \mathsf{v}_n] = \mathsf{E}_{MAL}[\![\mathsf{e}]\!]$$
$$\text{let } \Gamma_1, cs_1 = \mathsf{S}[\![\mathsf{s}_1]\!](\Gamma)$$
$$\text{let } cs'_1 = \mathsf{v}_1\ \mathsf{as\ string} +_v cs_1$$
$$\vdots$$
$$\text{let } \Gamma_n, cs_n = \mathsf{S}[\![\mathsf{s}_1]\!](\Gamma)$$
$$\text{let } cs'_n = \mathsf{v}_1\ \mathsf{as\ string} +_v cs_n$$
$$\text{let } \Gamma, cs'_1 +_h \ldots +_h cs'_n$$

Figure 4. Semantics for debugging spreadsheets for a subset of MAL. For conciseness, the semantics does not contain expression inlining and caches for non-bound variables and data objects.

could evaluate the function application and include only the result in the spreadsheet, even though this approach introduces possible inconsistencies. To keep things simple, we use the following prioritised strategy: 1) look for a spreadsheet counterpart and 2) let MAL handle the evaluation.

7.3 Semantics of Debugging Spreadsheets

We present a semantics for debugging spreadsheet for a subset of MAL's expressions and statements in Figure 4. While users will need a way of specifying what part of an execution they are interested in debugging, we will assume some appropriate mechanism (e.g., statements, command-line arguments, or breakpoint conditions) exists externally to the described semantics. A spreadsheet cell, $\mathsf{c} \in \mathsf{cell}$, and spreadsheet expressions, $\mathsf{e}_{ss} \in \mathsf{E}_{ss}$, can be understood intuitively and are similar to what is found in *Spreadsheets implementation technology* [27]. We use `[]` to denote the empty cell, which is used only for the purpose of layout. A block of cells, $\mathsf{cs} \in \mathsf{cell}[,]$, spans a rectangle. We consider a single cell as a singleton block of cells. A block of cells may be row-column indexed, e.g., `cs[1,0]`. Two blocks may be composed either horizontally or vertically with the left-associative operators $+_h$ and $+_v$, respectively, with blocks aligned at the top and left, respectively. We define the function `label` that creates a cell block that consists of an expression and a label above it.

$$\mathsf{label} : \mathsf{E}_{\mathsf{ss}} * \mathsf{string} \to \mathsf{cell}[,]$$
$$\mathsf{label}(\mathsf{e}_{\mathsf{ss}}, \mathsf{l}) = \mathsf{TextCell\ l} +_{\mathsf{v}} \mathsf{Cell\ e}_{ss}$$

We allow ourselves to appeal to the actual MAL evaluation of expressions with the oracle function $\mathsf{E}_{MAL}[\![\mathsf{e}]\!] : \mathsf{E} \to \mathsf{E}_{ss}$ that takes a MAL expression and returns a spreadsheet value representing the evaluated expression. We assume that an evaluation engine exists that maintains relevant context. This trick allows us to present the debugging spreadsheet semantics without also having to present MAL's semantics. We wrap $\mathsf{E}_{MAL}[\![e]\!]$ in the function `MAL` that labels the resulting value with the original expression.

$$\mathsf{MAL} : \mathsf{E} \to \mathsf{cell}[,]$$
$$\mathsf{MAL(e)} = \mathsf{label}(\mathsf{E}_{\mathsf{MAL}}[\![\mathsf{e}]\!], \mathsf{e\ as\ string})$$

We use the environment $\Gamma \in \mathsf{env}$ to keep track of where local variables are placed in cells. Here `env` is of type $\mathsf{string} \to \mathsf{cell}$. The function $\mathsf{E}[\![\mathsf{e}]\!](\Gamma)$ takes the expression `e` in the environment Γ returns a block of cells with the result expression at the leftmost and second topmost cell, i.e., at index 1,0. Likewise, the function $\mathsf{S}[\![\mathsf{s}]\!](\Gamma)$ takes the expression `s` in the environment Γ and returns a block of cells and an updated environment.

8 Lessons Learned

When reflecting on what we learned from the workshop, we move from the perspective of MAL's design to that of

DSL co-design in general and then discuss potential problems with transferring our experiences to other situations by discussing internal and external threats to validity. We take the methodological standpoint that an in-depth case study does provide grounds for generalisations [15]. This standpoint is the reason why we need to discuss the specifics of our workshop since it lets other practitioners thoroughly compare their design situation to ours and see whether our lessons learned are applicable to them.

From the perspective of the design of MAL, the workshop broadened our view of what quality assurance is in actuarial practices to also include analysis. Therefore, analytical tools are important for an in-depth understanding of the complicated mathematics modelled by management actions. If we are to support users in their work activities, we should both create tools that allow for strict test requirements to specific calculations and tools for analysing specific behaviour. The workshop led us to three concrete quality assurance measures supporting this workflow which we think will greatly improve MAL. Fragment debugging was already partly implemented, but the workshop emphasised the need to improve the technical solution both of the domain-specific and the general-purpose solution. In addition, we were happy to hear that users during the workshop pointed to problems that MAL in itself seeks to solve. MAL both seeks to improve program understandability as requested by users and eliminate the need for users to worry about the projection engine.

From the perspective of DSL co-design, it is possible to actively engage non-programming experts in the design of quality assurance. Experts may have another perspective on quality assurance, but if everyone is flexible in their definitions, discussion can be fruitful. These kinds of differences can make it challenging to prepare a precise plan for the workshop. Even if fallback plans are prepared, the workshop facilitator should be open to changes if the workshop activities reveal such a need. Still, we believe it is important for the facilitator to structure the workshop and prepare discussion inputs since workshop participants cannot be expected to generate designs on their own. We believe that the design of *debugging spreadsheets* can be used to debug other DSLs in similar domains.

For internal threats to validity, it is possible that other processes could have led us to change our perspective on quality assurance in MAL with similar quality assurance measures. While it is difficult to mitigate such a threat, we note that our prior work with designing MAL did not lead to such a shift in perspective. Also, we had a selection bias in workshop participants since all participants volunteered to participate, which means they may not represent the general domain expert who may be more reluctant to engage in workshop activities. However, we find it to be a reasonable necessity that all participants should willingly participate and engage in a workshop for it to be successful.

For external threats to validity, we could have benefited from having experts from a mathematical domain with a vocabulary somewhat close to that of software engineering, meaning our experience are not transferable to non-mathematical domains. Second, one could fear that users from different companies were unwilling to share potential business secrets. Such fear did not seem to limit our participants. We primarily attribute this willingness to participants sharing an interest in improving the projection platform and to a high level of mutual trust between Danish actuaries. Third, one could fear that the design workshop could strain customer relations and become an arena for contract negotiations.

9 Conclusion

In this paper, we have investigated the possibility of using co-design workshops to design DSL quality assurance measures with non-programming experts. We have done so to mitigate the risk of designing traditional software engineering quality assurance measures that are only partly usable in the domain. We first gave a short presentation of how MAL can be used in asset/liability projections. Then we described our workshop plan and experiences with executing the plan with prospective users of MAL. One result was that actuaries, and likely other non-programming experts, care deeply about quality assurance and can participate generatively in co-design workshops. Another result of the workshop was that our focus shifted from testing tools to analytical tools as quality assurance measures. We consider this shift in itself as a sign of the workshop being productive for the design project. We believe that our approach to co-designing quality assurance may be used by others facing the similar challenge of designing measures for non-programming experts. In addition, we have shown how the workshop led to three concrete quality assurance measures. We believe that our findings regarding quality assurance can influence the design of further DSLs and, especially *debugging spreadsheets* can be applied to other domains with heavy usage of spreadsheet calculations.

Acknowledgments

We thank Innovation Fund Denmark (7076-00029B), Edlund, and their customers for their participation.

References

[1] Robin Abraham and Martin Erwig. 2007. UCheck: A spreadsheet type checker for end users. *Journal of Visual Languages & Computing* 18, 1 (Feb. 2007), 71–95. https://doi.org/10.1016/j.jvlc.2006.06.001

[2] Rui Abreu, André Riboira, and Franz Wotawa. 2012. Debugging Spreadsheets: A CSP-based Approach. In *2012 IEEE 23rd International*

Symposium on Software Reliability Engineering Workshops. 159–164. https://doi.org/10.1109/ISSREW.2012.31

[3] Erwin Aertbeliën and Joris De Schutter. 2014. eTaSL/eTC: A constraint-based task specification language and robot controller using expression graphs. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 1540–1546. https://doi.org/10.1109/IROS.2014.6942760 ISSN: 2153-0866.

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-37521-7.

[5] Earl T. Barr and Mark Marron. 2014. Tardis: affordable time-travel debugging in managed runtimes. *ACM SIGPLAN Notices* 49, 10 (Oct. 2014), 67–82. https://doi.org/10.1145/2660193.2660209

[6] Kerl Bodker, Finn Kensing, and Jesper Simonsen. 2004. *Participatory It Design: Designing for Business and Workplace Realities*. MIT Press, Cambridge, MA, USA. https://doi.org/10.1109/TPC.2005.853942.

[7] Holger Stadel Borum, Malthe Ettrup Kirkbro, and Peter Sestoft. 2018. *Spreadsheet Patents*. Technical Report TR-2018-200.

[8] Holger Stadel Borum, Henning Niss, and Peter Sestoft. 2021. On Designing Applied DSLs for Non-programming Experts in Evolving Domains. In *Proceedings of the 24th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS '21)*. Association for Computing Machinery, Virtual Event, Fukuoka. (To be published).

[9] Ingrid Burkett. 2012. An introduction to co-design. (2012).

[10] Hyun Cho, Jeff Gray, and Eugene Syriani. 2012. Creating visual Domain-Specific Modeling Languages from end-user demonstration. In *2012 4th International Workshop on Modeling in Software Engineering (MISE)*. 22–28. https://doi.org/10.1109/MISE.2012.6226010 ISSN: 2156-7891.

[11] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[12] Bas Cornelissen, Andy Zaidman, Arie van Deursen, and Bart van Rompaey. 2009. Trace visualization for program comprehension: A controlled experiment. In *2009 IEEE 17th International Conference on Program Comprehension*. 100–109. https://doi.org/10.1109/ICPC.2009.5090033 ISSN: 1092-8138.

[13] J. Steve Davis. 1996. Tools for spreadsheet auditing. *International Journal of Human-Computer Studies* 45, 4 (Oct. 1996), 429–442. https://doi.org/10.1006/ijhc.1996.0061

[14] Michael De Rosa, Jason Campbell, Padmanabhan Pillai, Seth Goldstein, Peter Lee, and Todd Mowry. 2007. Distributed Watchpoints: Debugging Large Multi-Robot Systems. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*. IEEE, Rome, Italy, 3723–3729. https://doi.org/10.1109/ROBOT.2007.364049 ISSN: 1050-4729.

[15] Bent Flyvbjerg. 2006. Five Misunderstandings About Case-Study Research. *Qualitative Inquiry* 12, 2 (April 2006), 219–245. https://doi.org/10.1177/1077800405284363 Publisher: SAGE Publications Inc.

[16] Andy Gordon and Simon Peyton Jones. 2021. Enriching Excel with higher-order functional programming. https://www.microsoft.com/en-us/research/blog/lambda-the-ultimatae-excel-worksheet-function/

[17] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2012. Community-driven language development. In *2012 4th International Workshop on Modeling in Software Engineering (MISE)*. 29–35. https://doi.org/10.1109/MISE.2012.6226011 ISSN: 2156-7891.

[18] Javier Luis Cánovas Izquierdo, Jordi Cabot, Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2013. Engaging End-Users in the Collaborative Development of Domain-Specific Modelling Languages. In *Cooperative Design, Visualization, and Engineering (Lecture Notes in Computer Science)*, Yuhua Luo (Ed.). Springer, Berlin, Heidelberg, 101–110. https://doi.org/10.1007/978-3-642-40840-3_16

[19] Birthe Merethe Jensen, Martin Dencker Raffnsøe, and Jingyu She. 2019. Forsikrings- og pensionssektoren i ny kvartalsvis statistik.

[20] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. 2003. A user-centred approach to functions in Excel. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming (ICFP '03)*. Association for Computing Machinery, Uppsala, Sweden, 165–176. https://doi.org/10.1145/944705.944721

[21] Marco Kuhrmann. 2011. User Assistance during Domain-specific Language Design. *FlexiTools Workshop* (2011).

[22] Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2015. Example-driven meta-model development. *Software & Systems Modeling* 14, 4 (Oct. 2015), 1323–1347. https://doi.org/10.1007/s10270-013-0392-y

[23] A.D. Malony, D.H. Hammerslag, and D.J. Jablonowski. 1991. Traceview: a trace visualization tool. *IEEE Software* 8, 5 (Sept. 1991), 19–28. https://doi.org/10.1109/52.84213 Conference Name: IEEE Software.

[24] Silvia Mirri, Marco Roccetti, and Paola Salomoni. 2018. Collaborative design of software applications: the role of users. *Human-centric Computing and Information Sciences* 8, 1 (March 2018), 6. https://doi.org/10.1186/s13673-018-0129-6

[25] Brad A. Myers, John F. Pane, and Andy Ko. 2004. Natural programming languages and environments. *Commun. ACM* 47, 9 (Sept. 2004), 47–52. https://doi.org/10.1145/1015864.1015888

[26] Elizabeth B.-N. Sanders and Pieter Jan Stappers. 2008. Co-creation and the new landscapes of design. *CoDesign* 4, 1 (March 2008), 5–18. https://doi.org/10.1080/15710880701875068

[27] Peter Sestoft. 2014. *Spreadsheet Implementation Technology: Basics and Extensions*. MIT Press, Cambridge, MA, USA. https://doi.org/10.7551/mitpress/8647.001.0001.

[28] J. Sienkiewicz and T. Radhakrishnan. 1996. DDB: a distributed debugger based on replay. In *Proceedings of 1996 IEEE Second International Conference on Algorithms and Architectures for Parallel Processing, ICA/sup 3/PP '96*. 487–494. https://doi.org/10.1109/ICAPP.1996.562913

[29] Jesús Sánchez-Cuadrado, Juan de Lara, and Esther Guerra. 2012. Bottom-Up Meta-Modelling: An Interactive Approach. In *Model Driven Engineering Languages and Systems (Lecture Notes in Computer Science)*, Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson (Eds.). Springer, Berlin, Heidelberg, 3–19. https://doi.org/10.1007/978-3-642-33666-9_2

[30] Daniel Strüber, Anthony Anjorin, and Thorsten Berger. 2020. Variability representations in class models: an empirical assessment. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS '20)*. Association for Computing Machinery, Virtual Event, Canada, 240–251. https://doi.org/10.1145/3365438.3410935

[31] T. Systa, Ping Yu, and H. Muller. 2000. Analyzing Java software by combining metrics and program visualization. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*. 199–208. https://doi.org/10.1109/CSMR.2000.827328

[32] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoît Combemale. 2013. Variability Support in Domain-Specific Language Development. In *Software Language Engineering (Lecture Notes in Computer Science)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer International Publishing, Cham, 76–95. https://doi.org/10.1007/978-3-319-02654-1_5

[33] Maria Jose Villanueva, Francisco Valverde, and Oscar Pastor. 2014. Involving End-Users in the Design of a Domain-Specific Language for the Genetic Domain. In *Information System Development*, María José Escalona, Gustavo Aragón, Henry Linger, Michael Lang, Chris Barry, and Christoph Schneider (Eds.). Springer International Publishing, Cham, 99–110. https://doi.org/10.1007/978-3-319-07215-9_8

Appendix C

Transforming Domain Models to Efficient C# for the Pension Industry

Transforming Domain Models to Efficient C# for the Danish Pension Industry

HOLGER STADEL BORUM, IT University of Copenhagen, Denmark
MORTEN TYCHSEN CLAUSEN, IT University of Copenhagen, Denmark

Danish insurance and pension companies are required by financial regulations to report certain financial quantities to prove that they are solvent and managed responsibly. Parts of these quantities are computed the same way for all companies, whereas so-called management actions, describing, e.g., surplus sharing, vary between companies. Hence it is desirable to have a flexible calculation platform that allows actuaries to easily create company-specific models, which are also computationally efficient. In this paper, we present our work with implementing a code generator for a DSL called the Management Action Language (MAL) as a form of variability management. While one of the goals of MAL is to generate efficient code from an actuary's specification, it is non-trivial how to produce such code. We identify four reoccurring patterns in the models created by actuaries as subjects to optimisations. We describe our process for implementing a code-generator by a) identifying four specification patterns (inheritance, union types, type filtering, and numerical maps) that are pervasive in these calculations, and b) describing how to generate efficient C# from MAL for these patterns. We evaluate the code-generator by benchmarking it against handwritten production code and show an approximate 1.3× speedup in a production environment. This evaluation demonstrates that, with MAL, an individual pension company may reuse the general calculation platform and all of the optimisations built into MAL's code generator when modelling the company's business rules.

CCS Concepts: • **Software and its engineering → Source code generation**; **Domain specific languages**; • **General and reference → Performance**.

Additional Key Words and Phrases: Domain specific languages, performance, code generation, vernacular software development

ACM Reference Format:
Holger Stadel Borum and Morten Tychsen Clausen. 2022. Transforming Domain Models to Efficient C# for the Danish Pension Industry. In *Proceedings of Modeling Language Engineering (MLE'22).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Current legislation requires Danish insurance and pension companies to report a number of quantities as a way of proving that they are solvent and managed responsibly [8]. The calculations of these quantities are complicated since they a) involve projecting financial products into the future, b) must model a specific company, c) are calculated in different economic scenarios, and d) involve complex mathematical formulas. While parts of these calculations are identical for all companies (such as the projection of assets and reserves), other parts are company-specific since they model company-specific rules called management actions (such as surplus sharing). This duality means that a projection platform consists of a generic company-independent part and a specialised company-specific part, as shown in Figure 1.

In the current practice, actuaries submit management action specifications to a projection platform written in a general-purpose language such as C#, as seen in Figure 1. This practice is challenged by actuaries with limited

Manuscript submitted to ACM

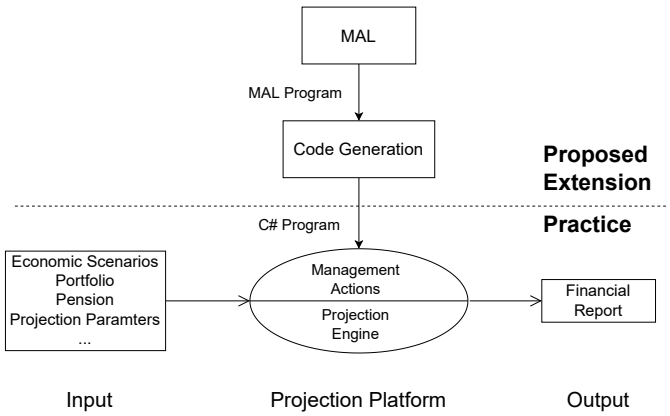

Fig. 1. A diagram with the current projection process where the projection platform is separated into a general *projection engine* and company-specific *management actions*. In the proposed extension, the management actions are generated from a specification written in the Management Action Language.

programming experience (or *vernacular software developers* [20]) being required to specify *efficient* management actions due to projections being computationally expensive.

Our proposal is to modify the current practice with a domain-specific language (DSL) and a code generator that generates efficient management actions from more declarative specifications. Our DSL called the Management Action Language (MAL, Section 2), was designed for the purpose of allowing actuaries as vernacular software developers to model management actions and have them automatically transformed, optimised, and executed efficiently on a projection platform. In this paper, we present our work with implementing a code generator for MAL to perform this transformation (Section 3). We identify four specification patterns that find are pervasive in management action specifications and discuss how we generate efficient code for these in a target language with a more restrictive type system (Section 3.1-3.3).

In doing so, we demonstrate a case where a DSL can be used to generate code that is 1.3× more efficient compared to handwritten production code in the target language (Section 4).

In short, the contributions of this paper are:

- An identification of specification patterns pervasive in management action specifications.
- Implementation strategy for generating code from these specifications with C# as target language.
- Positive results that show a speedup when benchmarking generated code against comparable handwritten code.
- Negative results that do not show a speedup for loop vectorisation.

2 THE DANISH PENSION INDUSTRY AND DSLS

The Danish pension and life insurance industry manages a large number of assets [12], and it is, therefore, important for the Danish economy that the industry is healthy. Here we give an ultra-compact conceptual model of the domain that is coherent with the DSL concepts described in the paper, primarily based on [17]. A *pension* (or *life insurance*) is a saving to be paid to the *policyholder* (or a beneficiary) upon the policyholder reaching a specific age (or dying). A *pension company* is a company that offers pension and life insurance *products* to customers. A *policy* is an

instance of a product bought by a specific customer. The policies held by a company are, collectively, the *portfolio* of the company. The *reserve* of a policy is its present value which is the total of all its future payments. A company is *solvent* when it can meet all expected future insurance claims. A company determines whether it is solvent by computing its *balance* consisting of its *assets*, e.g., bonds, and *liabilities*, where the reserve of policies is a substantial part of the liabilities. It is common to put a policy into different types of *groups* according to its *interest* rate, biometric *risk*, and administrative *expense* profile.

The projection of the balance is one way for a pension company to argue that it is responsibly managed and will remain solvent. A pension company is required by Danish regulation to make such an argument to the Danish Financial Supervisory Authority (FSA). There are two major challenges in performing such a projection which we call the *performance* challenge and *variability* challenge. The performance challenge is that such projections are computationally expensive. A projection should take many different economic scenarios (in the order of 10 up to 1000) into account to simulate how the portfolio behaves with varying yields and losses on investments. One way of doing so is to parametrise a projection on an economic scenario and then perform a Monte Carlo simulation over different economic scenarios. This simulation approach is the one used by the projection platform that MAL was designed for. The variability challenge is that a projection must model certain business rules of the company that specify how investment yields and losses are generally handled. These business rules are called management actions. Management actions are not used only as input to a balance projection but must also be shown and approved by the board of directors of a pension company and be reported with "recipe-like" precision to the FSA.

The Management Action Language was designed to allow actuaries to model management action in balance projections. A design goal was to emphasise the mathematics of these models while minimising boilerplate code. MAL consists of three parts: On top, a module system lets actuaries group management actions into meaningful computational units. Each module consists of two parts: First, data declarations and data contracts define what data a module requires and provides (see Figure 3 for a small example of data declarations). Second, imperative management actions and pure functional expressions define how a module computes data. Figure 2 shows a fragment of a MAL module that uses computational patterns that are pervasive when modelling management actions.

```
update eGroup in Groups:Expense {
  let reserveInclIRTA = eGroup.Reserve + eGroup.PAL.InvestmentReturnTaxAsset
  let periodTechnicalExpenses =
    sum( map p1 in eGroup.Policies:OneStatePolicy {
           Util.SumExpenses(p1.Result.PeriodTechnicalExpenses)
         }
       )
  eGroup.ExpenseDividend =
      if periodTechnicalExpenses == 0
      then 0
      else bound(0.02 * reserveInclIRTA / (periodTechnicalExpenses / Projection.PeriodLength), 0, 1)
}
```

Fig. 2. An abridged but realistic MAL fragment demonstrates commonly used patterns, namely: a) inheritance based type filtering of Expense and `OneStatePolicy` and b) a often used map-sum pattern.

3 CODE GENERATION

In this section, we give a high-level description of our approach to translating MAL into C#. The translation is subject to the following somewhat conflicting design considerations.

```
data Group {
  Child : Group
}
data Risk extends Group {
  Child : Risk
}
data Expense extends Group {
}
```

```
public class Risk : Group,Expense_Risk {
  private Risk _Child;
  public new Risk Child
  { get {return _Child;}
    set {_Child = value;
         base.Child = value;}
  }
}
```

```
public interface Expense_Risk {
  Group Child { get; set; }
}
```

```
public class TypeSpan_Expense_Risk : IEnumerable<Expense_Risk> {
  private Expense[] Expense;
  private Risk[] Risk;
  public TypeSpan_Expense Filter_Expense() {
    return new TypeSpan_Expense(Expense);
  }
  /* Remaining Filters and enumeration omitted. */
}
```

Fig. 3. Examples of code generation. **Top left:** An artifical MAL-snippet with a data declaration for `Group` and the two subtypes `Risk` and Expense. **Top Right:** Generated C# class for `Risk`. **Middle:** Generated C# interface for the Expense and `Risk` union type. **Bottom:** The generated C# TypeSpan that is a collection type for Expense and `Risk` with constant time filtering.

(1) C# was chosen as the target language for the code generator due to a wish to potentially generate a self-contained C# version of a MAL program.
(2) The produced C# should be compatible with several existing interfaces and flexible enough to handle some evolution of these.
(3) Performance of the generated code was important due to the computational cost of calculations.

During the design of MAL, we identified four specification patterns (inheritance, union types, type filtering, and numerical maps) that were pervasive in expressing management actions. As an example, there may exist three different kinds of Groups (see Section 2), say Risk, Interest, and Expense (object-oriented inheritance). Both Risk and Expense groups may have a shared quantity we want to compute (union type). We may compute the value by choosing all Risk or Expense groups (type filtering) and then, for each group, compute the value specified by summing over a map of the group's policies (numerical map). We describe how we generate C# code for each of the above-specified patterns. To accommodate type filtering, we generate what we call a *TypeSpan*, which is a collection of objects with constant time type filtering. In Section 4, we argue the generated code is comparatively efficient and correct.

Figure 3 shows an example of how we translate data declarations in MAL to classes and interfaces in C# using the subsequently presented translation schemes. However, to conserve space, we will primarily work on an abstract representation of a MAL program without presenting the full abstract or concrete syntax. To do so, given a MAL program, we define L to be a set of labels of data fields used, T to be the set of the program's types, $\mathsf{T}[\![\_]\!]$ to be a type translation from MAL to C#, and $<:$ to be a subtype relation on T with $\mathtt{lub}^{<:}(\_)$ as a function providing the least upper bound on a set of types. We use dom(_) to denote the domain of a function, $\rightharpoonup$ to denote a partial map, and $\mathbb{P}(\_)$ to denote the powerset.

$$\mathtt{D}[\![\mathtt{d}]\!] = \boxed{\begin{array}{l}\texttt{public class } \boxed{\texttt{d}} \ \boxed{\texttt{: d' when } \exists(\texttt{d},\texttt{d'}) \in \texttt{I}} \\ \{ \\ \quad \boxed{\mathtt{Fields}_d[\![\mathtt{dom(F(d))}]\!]} \\ \}\end{array}}$$

$$\mathtt{Fields}_d[\![\{\mathtt{lb}_1, ..., \mathtt{lb}_n\}]\!] = \boxed{\mathtt{Field}_d[\![\mathtt{lb}_1]\!]} \ ... \ \boxed{\mathtt{Field}_d[\![\mathtt{lb}_n]\!]}$$

$$\mathtt{Field}_d[\![\mathtt{lb}]\!] = \begin{cases} \boxed{\begin{array}{l}\texttt{private } \boxed{\mathtt{T}[\![\mathtt{F(d)(lb)}]\!]} \ \_\boxed{\texttt{lb}}; \\ \texttt{public new } \boxed{\mathtt{T}[\![\mathtt{F(d)(lb)}]\!]} \ \boxed{\texttt{lb}}; \\ \{ \texttt{ get } \{\texttt{return } \_\boxed{\texttt{lb}};\} \\ \quad \texttt{set } \{\_\boxed{\texttt{lb}} \texttt{ = value;} \\ \qquad \texttt{base.}\boxed{\texttt{lb}} \texttt{ = value}\} \\ \}\end{array}} & \text{if } \exists\, d' \,.\, (d,d') \in \mathtt{I} \\ & \quad \wedge\, lb \in dom(\mathtt{F}(d')) \\ \boxed{\texttt{public } \boxed{\mathtt{T}[\![\mathtt{F(d)(lb)}]\!]} \ \boxed{\texttt{lb}};} & \text{Otherwise} \end{cases}$$

Fig. 4. Translation scheme, $\mathtt{D}[\![\_]\!]$, for generating classes from data declarations. The scheme uses $\mathtt{Fields}_d[\![\_]\!]$ to generate a field for all labels (lb) of a data declaration using $\mathtt{Field}_d[\![\_]\!]$. Notice how $\mathtt{Field}[\![\_]\!]$ shadows a field when a data declaration specialises the type of the field. **Legend:** A solid frame $\boxed{\ }$ delimits a quasi-quotation area where we may make use of our functions, variables, and some pseudocode. A dashed frame $\boxed{\ }$ delimits code parts that are only included when relevant.

3.1 Data Declarations and Inheritance

MAL implements a single inheritance system that affords users the possibility of describing different versions of the same actuarial concept. That is, if a user wants to create a Risk group and an Expense group, they can do so by letting both inherit from Group.

Definition 1. The data declarations of a MAL program can be represented by the tuple $(\mathtt{D}, \mathtt{I}, \mathtt{F})$ where:

- $\mathtt{D}$ is a set of data declaration names.
- $\mathtt{I}$ is a binary relation on $(d_1, d_2) \in \mathtt{D}$ denoting that d_1 inherits from d_2. $\mathtt{I}^*$ denotes the transitive closure on $\mathtt{I}$.
- $\mathtt{F} : \mathtt{D} \rightarrow \mathtt{L} \rightharpoonup \mathtt{T}$ is a map from a data declaration name to its fields to their type.

We only consider well-formed data declaration where (1) a data declaration at most inherits from a single other declaration and (2) an inheritance only specialises fields from its supertype while respecting the subtype relation, or formally:

- $\forall (d_1, d_2) \in \mathtt{I} \,.\, \forall d_3 \in \mathtt{D} \,.\, d_2 \neq d_3 \Rightarrow (d_1, d_3) \notin \mathtt{I}$
- $\forall (d_1, d_2) \in \mathtt{I} \,.\, \forall lb \in \mathrm{dom}(\mathtt{F}(d_2)) \Rightarrow \mathtt{F}(d_1)(lb) <: \mathtt{F}(d_2)(lb)$

Figure 4 shows a scheme $\mathtt{D}[\![\_]\!]$ for translating data declarations into C#. The primary thing to notice about the scheme is that when a data declaration is extended with a field that is a subtype of the original field, then this field is shadowed in the generated subclass. This choice means that in MAL, there are severe restrictions to field assignments on a superclass to ensure that the generated class hierarchy remains consistent throughout an execution.

3.2 Union Types and TypeSpans

To provide users with the possibility of working flexibly with collections of entities, MAL implements data declaration filtering on collections with the possibility of doing so with a union type.

Definition 2. The union types used in a MAL program are denoted by $\mathtt{U} \subseteq \mathbb{P}(\mathtt{D})$. We assume we have an injective function $id_{\mathtt{U}} : \mathbb{P}(\mathtt{D}) \to \mathtt{String}$ that provides a union type with a unique name. The function $\mathtt{F}_{\mathtt{U}} : \mathbb{P}(\mathtt{D}) \to \mathtt{L} \rightharpoonup \mathtt{T}$ provides types to the fields of $u : \mathtt{U}$ with $\mathtt{F}_{\mathtt{U}}(u)(l) = \text{lub}^{<:}(\{\mathtt{F}(d)(l) \mid d \in u\})$. The function $\mathtt{F}_{\mathtt{U}}(u)$ is only defined on the domain $\cap_{d \in u} \text{dom}(\mathtt{F}(d))$.

A union type $u : \mathtt{U}$ is well-formed when it (1) is non-empty and (2) all names within the union inherit transitively from a given data declaration, or formally:

- $u \neq \emptyset$
- $\exists d' \in \mathtt{D} \,.\, \forall d \in u \,.\, (d, d') \in \mathtt{I}^*$

For each union type of a MAL program, we generate an interface corresponding to the type (Figure 5) and a TypeSpan that serves as a collection for the union type with constant time filter operations (Figure 5). While we potentially need to make this generation for exponentially many combinations of a data declaration's subtypes, we can limit this number to the union types that are actually used in a concrete MAL program. This means that while the size of the generation is theoretically upper bounded by an exponential function, it is linear in the length of a program. The generation of TypeSpan provides users with a constant filter operation on a collection of data declarations, with zero cost for type casts or type checking.

$\mathtt{U}[\![\mathtt{u}]\!] =$

```
public interface [id_U(u)] {
  [UFields_u[[dom(F(u))]]]
}
```

$\mathtt{UFields}_u[\![\{\mathtt{lb}_1, ..., \mathtt{lb}_n\}]\!] =$

```
[T[[F_u(u)(lb_1)]]] [lb_1] { get; set; }
... public [T[[F_u(u)(lb_n)]]] [lb_n] { get; set; }
```

$\mathtt{TS}[\![\{\mathtt{d}_1, ..., \mathtt{d}_n\}]\!] =$

```
public class TypeSpan_[id_u({d_1, ..., d_n})]
             : IEnumerable<[id_u({d_1, ..., d_n})]> {
  private [T[[d_1]]][] [d_1];
  ... private [T[[d_n]]][] [d_n];

  [For all u' ∈ P({d_1, ..., d_n}) generate Filter[[u']]]
  // constructors, enumeration, etc. omitted
}
```

$\mathtt{Filter}[\![\{\mathtt{d}_1, ..., \mathtt{d}_n\}]\!] =$

```
public TypeSpan_[id_U({d_1, ..., d_n})] Filter_[id_U({d_1, ..., d_n})]() {
  return new TypeSpan_[id_U({d_1, ..., d_n})](d_1, ..., d_n);
}
```

Fig. 5. **On top:** Translation scheme, $\mathtt{U}[\![\_]\!]$, for generating interfaces for union types. The scheme uses $\mathtt{UFS}_u[\![\_]\!]$ to generate the fields on the union type. **On bottom:** Translation scheme for generating the TypeSpan for a union type. Note the constant time filtering operations created by $\mathtt{Filter}[\![\_]\!]$.

3.3 Vectorisation

The expression language of MAL contains a map operation since it is frequently used in management specifications. When generating code for a numerical map operation, it seemed like an obvious optimisation to generate code using advanced vector extensions (AVX). We call this form of generation *vectorisation*.

Our approach to vectorisation was to vectorise all map operations that contained an arithmetic operation, which is always possible since MAL's expression language is free of side effects. Since most values are fields on an object in MAL, we had to transform relevant values into AVX vectors when needed. We used the following high-level process of vectorisation:

(1) Identify that a map operation uses an arithmetic operation.
(2) Traverse the collection being mapped over in slices of the hardware-specific size of the AVX registers.
(3) Transform relevant data to AVX vectors.
(4) Use AVX instructions whenever possible.

However, in an actual implementation, several other details show up to ensure efficient code, such as an identification of loop-constant expressions, removal of intermediate arrays, compatibility with deforestation [25], and improving heuristics for choosing when to vectorise an expression.

The idea behind the optimisation is that an AVX instruction is used on $2 * \texttt{vl}$ elements rather than just 2 at a time, where vl is the hardware-specific register size. The total number of operations performed should thus become $\sim\frac{1}{\texttt{vl}}$ of the number of operations in a non-vectorised translation.

4 EVALUATION

We evaluate MAL's implementation from a technical perspective since a user-oriented evaluation has previously been conducted [7]. We evaluate by comparing the performance of generated code against an equivalent handwritten specification written in C#. This specification is realistic in the sense that significant effort has been put into its over 14,000 lines of code while being developed with the entire projection platform. It is also realistic since it serves as a template implementation for several pension companies who may either modify the template or write their own implementation from scratch.

4.1 Performance Experiments

We are interested in investigating how the performance of code generated from MAL compares to comparative handwritten C# code. For performance measures, two different benchmark setups are used. The first local setup almost executes only management code on a dedicated benchmark machine to investigate how the generated code performs. The second production setup executes a full projection in a realistic execution on a cloud service to investigate the total impact of using MAL in the real system.

4.1.1 Only management code. We are interested in a) how does MAL perform on a portfolio of realistic size? And b) how does the MAL implementation scale on increasing portfolio sizes? The benchmark was performed on a machine with an Intel Xeon E5-2680 v3 with 48 logical 2.5 GHz cores and 32 GB of memory, running 64-bit Windows 10, version 20H2, and .NET 4.8.4084. The result of the benchmarks is seen in Figure 6.

Observation 1. The execution time of all solutions scales linearly in the number of policies.

We explain this observation from the fact that all solutions perform their majority of work linearly in the number of exercises.

Observation 2. For all number of policies, the solution generated from a MAL program is faster than the C# solution with a speedup factor of approximately 1.3×.

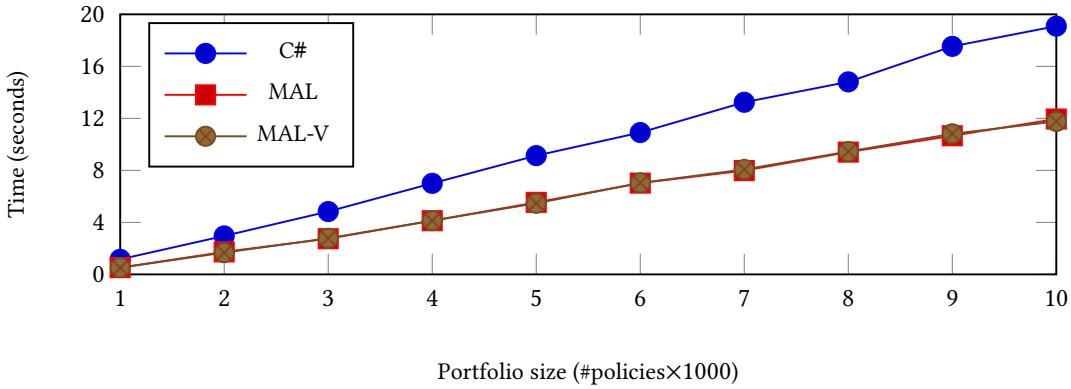

Fig. 6. Running time of benchmarking management code performed on randomly generated portfolios. The benchmarks were generated using BenchmarkDotNet [2] with `WarmupCount=8, IterationCount=40, InvocationCount=1`. One standard deviation is plotted as the error but it is not visible since it is generally $< 3\%$.

We explain this speedup from the second observation from the fact that: a) MAL's translation from data declarations to classes provides a pretty efficient memory layout compared to the C# solution, which often uses dictionaries and indirect lookups for similar purposes. b) MAL solution generates less garbage due to the use of classes and deforestation. c) MAL's easily reusable constant time type filtering without type casts is faster than checking every individual of a collection. Although this functionality is possible to write in C# it is difficult to do in any reusable manner without some form of code generation.

Observation 3. We cannot find any difference in the execution time of code generated with and without vectorisation.

We explain this negative result from the overhead of taking numerical values from an object and placing them into an array. We hypothesise that we need a more vectorisation-friendly memory layout to see a speedup.

4.1.2 Full projection. We are interested in how using MAL affects the part of the execution that does not involve executing management code directly. However unlikely, it is possible that MAL's memory footprint, some delayed execution, or slower serialisation means that the full projection using MAL could be slower, even if the management code executes faster. For full projections, it is worth noting that even though the majority of execution time takes place outside of management code, then the time spent in management has grown significantly (from around 10% to around 20%). It seems like the trend is likely to continue for two reasons. First, parts of the non-management code may be optimised for all projections (e.g., by solving some partial-differential equitations analytically instead of numerically). Second, new requirements to the projection platform have a tendency to be implemented as management code that is available for users to modify. Table 1 shows the speedup from using MAL in a production setting. From this benchmark, we conclude that MAL provides around 1.3× speedup compared to handwritten management actions and that MAL does not negatively affect the remainder of the projection. The setup and data used for this benchmark are quite different from that used in Figure 6, which means that their real times are not comparable.

Table 1. Benchmarks for full projections in a production environment on *Standard_F2s_v2* machines. The table shows two different projection setups with speedups for only management code and for the full projection.

	10k policies, 1 economic scenario			1k policies, 1k economic scenarios		
Task	C#	MAL	Speedup	C#	MAL	Speedup
Management Init	1.0 s	3.8 s	0.2×	238.8 s	469.9 s	0.51×
Management Update	26.4 s	17.3 s	1.77×	3198.0 s	1833.7 s	1.74×
Management Finalize	1.5 s	1.7 s	1.15×	123.3 s	131.5 s	0.94×
Management Total	**28.9 s**	**22.8 s**	**1.27×**	**3,560 s**	**2,435 s**	**1.46×**
Full Projection Total	**120.3 s**	**109.8 s**	**1.09×**	**15,943 s**	**14,486 s**	**1.10×**

4.2 Implementation Correctness

We argue that MAL's code generator is correct by passing a test suite that consists of the following four kinds of tests:

(1) MAL passes 9 relevant regression tests designed for the C#-solution. The only test that does not pass requires some extra functionality that is currently not present in MAL.
(2) MAL produces the same results as the code for C# randomly generated portfolios.
(3) Using FSCheck [1] to create a generator for random valid MAL programs, we have performed a series of property-based tests. The properties we tested for are:
   - Printing and then parsing a MAL program results in a syntactically equivalent program.
   - A typed MAL program retains its types after printing and parsing it.
   - Transforming an error-free MAL program produces an error-free C# program.
(4) A small suite of unit tests ensures the correctness of different smaller components.

5 RELATED WORK

Danish pension companies must perform balance projections based on management actions, and therefore these companies all need an executable formalisation of their management actions, whatever platform they use. While it is common for actuaries at Danish pension companies to publish their actuarial models for academic inspection and discussion, it is not common for Danish pension companies to publish information on their software solutions. However, from participating in project advisory board meetings and from an industry conference, we find that Danish pension companies specify their management actions in general-purpose languages (C# and Visual Basic) but are interested in other approaches. To our knowledge, our project is the only DSL that lets actuaries express management actions, and this article is the first on the efficient execution of management actions.

Domain-specific languages have been used several times in financial domains to model financial products or entities, e.g., [18, 22–24]. MAL is a spiritual successor to the Actuarial Modeling Language (AML) [9] that is used to model pension products. MAL is somewhat different from these DSLs since it does not seek to precisely model financial products but rather the management or manipulation of financial products. In this aspect, MAL is similar to the proprietary actuarial DSL T# [3], which is part of the company RPC's financial modelling platform called Tyche. While there is little public documentation of T#, the language approach is different from ours since it is a scripting language

that seemingly calls into an external library.

Other DSLs from vastly different domains with different target languages. For example, OptiML is designed to let machine learning researchers specify machine learning algorithms that are translated to efficient parallel code [21], SARVAVID is designed to ease genomic analysis by providing kernels for doing so and generating efficient C++ [15], and CFDLang is designed for specifying performance-critical operations in computational fluid dynamics can be translated into efficient C code [13, 19].

Many aspects used in MAL's code-generation design are covered generally and in-depth by others. For object-oriented inheritance, see [4], for union types see [11], and for vectorisation, see [14]. We considered looking into optimisations for loop parallelism [16] and data flattening [5] similar to the parallel language of Futhark [10], but found a conflict in the design of MAL compared to Futhark, which hindered such an approach. Futhark is a functional language where programs are built from parallel primitives operating on arrays of data. MAL is designed with actuaries as target users and leans toward object-oriented programming to be more reminiscent of C#, as actuaries in the Danish industry are most likely to have some limited experience with C# or Java. Users of MAL are thus not limited to programming using parallel building blocks, and any guarantee of performance increase from automatic parallelisation is lost.

TypeScript is a widely-used general-purpose language that implements the concept of union types [6], but with JavaScript as its target language, union types can seemingly just be erased when translating to JavaScript. Our work is different since we have to embed union types in the type system of C#.

6 CONCLUSION

In this paper, we have presented our work with generating efficient management actions for the pension industry from MAL. We identified four specification patterns that we found pervasive in the specification of management actions, namely: inheritance, union types, type filtering, and numerical maps. We described our efforts with generating efficient C# code for these patterns by showing translation schemes. We have shown that the code generated from our translation schemes leads to an approximate 1.3× speedup and conjecture that a significant part of this comes from our work with interfaces for union types and a collection with a constant time type filtering. We also report a negative result: our effort to generate code for numerical maps using AVX instruction resulted in no speedup.

Acknowledgements. We thank Innovation Fund Denmark (7076-00029B) for funding this work. We also thank Peter Sestoft for supervising this work.

REFERENCES

[1] FsCheck: Random Testing for .NET. https://fscheck.github.io/FsCheck/. Accessed July 2022.
[2] BenchmarkDotNet, 2021. https://benchmarkdotnet.org/. Accessed Dec 2021.
[3] RPC Tyche - Tyche, 2021. https://www.rpc-tyche.com/products/tyche.html. Accessed Dec 2021.
[4] Abadi, M., and Cardelli, L. *A Theory of Objects.* Monographs in Computer Science. Springer New York, New York, NY, 1996.
[5] Bergstrom, L., Fluet, M., Rainey, M., Reppy, J., Rosen, S., and Shaw, A. Data-only flattening for nested data parallelism. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming* (Shenzhen, China, Feb. 2013), PPoPP '13, Association for Computing Machinery, pp. 81–92.
[6] Bierman, G., Abadi, M., and Torgersen, M. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming* (Berlin, Heidelberg, 2014), R. Jones, Ed., Lecture Notes in Computer Science, Springer, pp. 257–281.

[7] Borum, H. S., Niss, H., and Sestoft, P. On Designing Applied DSLs for Non-programming Experts in Evolving Domains. In *Proceedings of the 24th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (Virtual Event, Fukuoka, Oct. 2021), MODELS '21, Association for Computing Machinery.

[8] Bruhn, K., and Lollike, A. S. Retrospective reserves and bonus. *Scandinavian Actuarial Journal* (Aug. 2020), 1–19.

[9] Christiansen, D., Grue, K., Niss, H., Sestoft, P., and Sigtryggsson, K. S. An Actuarial Programming Language for Life Insurance and Pensions. In *Proceedings of 30th International Congress of Actuaries* (2013).

[10] Henriksen, T., Serup, N. G. W., Elsman, M., Henglein, F., and Oancea, C. E. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2017), PLDI 2017, ACM, pp. 556–571.

[11] Igarashi, A., and Nagira, H. Union types for object-oriented programming. In *Proceedings of the 2006 ACM symposium on Applied computing* (Dijon, France, Apr. 2006), SAC '06, Association for Computing Machinery, pp. 1435–1441.

[12] Jensen, B. M., Raffnsøe, M. D., and She, J. Forsikrings- og pensionssektoren i ny kvartalsvis statistik, 2019. (In English: The Insurance Sector and Pension Sector in New Quarterly Annualy Statistic).

[13] Karl Friebel, F. A., Soldavini, S., Hempel, G., Pilato, C., and Castrillon, J. From Domain-Specific Languages to Memory-Optimized Accelerators for Fluid Dynamics. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)* (Sept. 2021), pp. 759–766. ISSN: 2168-9253.

[14] Kennedy, K., and Allen, J. R. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[15] Mahadik, K., Wright, C., Zhang, J., Kulkarni, M., Bagchi, S., and Chaterji, S. SARVAVID: A Domain Specific Language for Developing Scalable Computational Genomics Applications. In *Proceedings of the 2016 International Conference on Supercomputing* (Istanbul, Turkey, June 2016), ICS '16, Association for Computing Machinery, pp. 1–12.

[16] Oancea, C. E., and Rauchwerger, L. Logical inference techniques for loop parallelization. *ACM SIGPLAN Notices 47*, 6 (June 2012), 509–520.

[17] Perdersen, L. R. *Grundlæggende firmapension*, 5 ed. Forsikringsakademiets Forlag, 2014. (In English: Foundation in Company Pensions).

[18] Peyton Jones, S., Eber, J.-M., and Seward, J. Composing contracts: an adventure in financial engineering (functional pearl). *ACM SIGPLAN Notices 35*, 9 (Sept. 2000), 280–292.

[19] Rink, N. A., Huismann, I., Susungi, A., Castrillon, J., Stiller, J., Fröhlich, J., and Tadonki, C. CFDlang: High-level code generation for high-order methods in fluid dynamics. In *Proceedings of the Real World Domain Specific Languages Workshop 2018* (Vienna, Austria, Feb. 2018), RWDSL2018, Association for Computing Machinery, pp. 1–10.

[20] Shaw, M. Myths and mythconceptions: what does it mean to be a programming language, anyhow? *Proceedings of the ACM on Programming Languages 4*, HOPL (Apr. 2022), 234:1–234:44.

[21] Sujeeth, A. K., Lee, H., Brown, K. J., Chafi, H., Wu, M., Atreya, A. R., Olukotun, K., Rompf, T., and Odersky, M. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning* (Bellevue, Washington, USA, June 2011), ICML'11, Omnipress, pp. 609–616.

[22] van Deursen, A. Domain-Specific Languages versus Object-Oriented Frameworks: A Financial Engineering Case Study, 1997.

[23] Voelter, M. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, Lexington, KY, Jan. 2013.

[24] Voelter, M., Koščejev, S., Riedel, M., Deitsch, A., and Hinkelmann, A. A Domain-Specific Language for Payroll Calculations: An Experience Report from DATEV. In *Domain-Specific Languages in Practice*, A. Bucchiarone, A. Cicchetti, F. Ciccozzi, and A. Pierantonio, Eds. Springer International Publishing, Cham, 2021, pp. 93–130.

[25] Wadler, P. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science 73*, 2 (Jan. 1988), 231–248.

Appendix D

Survey of Established Practices in the Life Cycle of Domain-Specific Languages

Survey of Established Practices in the Life Cycle of Domain-Specific Languages

Holger Stadel Borum
hstb@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

Christoph Seidl
chse@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

ABSTRACT

Domain-specific languages (DSLs) have demonstrated their usefulness within many domains such as finance, robotics, and telecommunication. This success has been exemplified by the publication of a wide range of articles regarding specific DSLs and their merits in terms of improved software quality, programmer efficiency, security, etc. However, there is little public information on what happens to these DSLs after they are developed and published. The lack of information makes it difficult for a DSL practitioner or tool creator to identify trends, current practices, and issues within the field. In this paper, we seek to establish the current state of a DSL's life cycle by analysing 30 questionnaire answers from DSL authors on the design and development, launch, evolution, and end of life of their DSL. On this empirical foundation, we make six recommendations to DSL practitioners, scholars, and tool creators on the subjects of user involvement in the design process, DSL evolution, and the end of life of DSLs.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**;
• **General and reference → Surveys and overviews**.

KEYWORDS

Domain-specific languages, Survey

ACM Reference Format:
Holger Stadel Borum and Christoph Seidl. 2018. Survey of Established Practices in the Life Cycle of Domain-Specific Languages. In *Proceedings of ACM / IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS) (MODELS '22).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Domain-specific languages (DSLs) have been established as a multi-faceted tool in software engineering that can be used to improve usability [17], performance [27], security [25], and code reuse [19]. From an academic standpoint, little is known about most DSLs' life cycle beyond initial publications that demonstrate their merits, whether they be innovations within the application domain [20], investigations and innovations of DSL technologies [40], or more processual design considerations [12]. Reports on the broader life cycle of DSLs are scarce, if not altogether lacking. While some experts may have enough experience creating DSLs to know the current state of practices, it should be available to any DSL practitioner and tool-creator. Kosar et al. find in their systematic mapping study that most primary DSL studies focus on domain analysis, design and implementation, with only 10 out of 390 investigating maintenance and validation [24]. We find that the articles closest to investigating the DSL life cycle typically fall into two categories. First, a retrospective analysis summarises lessons learned from years of working with DSLs [23, 41]. Second, DSL evolution may be investigated through an explicit process discussion of evolution reasons and results [18, 35, 37, 39]. While both types of investigations are valuable and should be encouraged, their scarcity, combined with their diverse and non-standardised reporting, makes it difficult to use them for establishing the current state of DSLs' life cycles.

In this paper, we set out to alleviate the lack of information on DSLs after their publication by empirically investigating the established practices in the life cycles of DSLs. Our purpose is to survey current established practices in DSLs' life cycles and, from this foundation, make recommendations for future research. We do so through a questionnaire sent to selected DSL authors on their DSLs' life cycle phases (design and development, launch, evolution, and end of life). The questionnaire has an effective response rate of 43% from DSL authors whose DSL appeared in eight different DSL corpora (Section 2). The questionnaire is especially focused on topics difficult to examine outside of self-report since other methods more easily investigate technical choices such as tooling, see [21]. This focus translates to questions on user perspectives in different life phases. In addition, the survey investigates other aspects such as development setting, causes of DSL evolution, and reflections on the impact of different initiatives (Section 3). Based on our analysis, we make six recommendations for the DSL field (Section 4). We discuss threats to the validity of our findings (Section 5) and the utility of our approach compared to similar research (Section 6). To conclude, we summarise our findings (Section 7).

The contributions of this paper are:

- The presentation of empirical data regarding the design and development, launch, evolution, and end of life of DSLs.
- An analysis of data establishing current practices in the life cycles of DSLs.
- Empirically based recommendations for DSL practitioners.

MODELS '22, October 23–28, 2022, Montreal, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

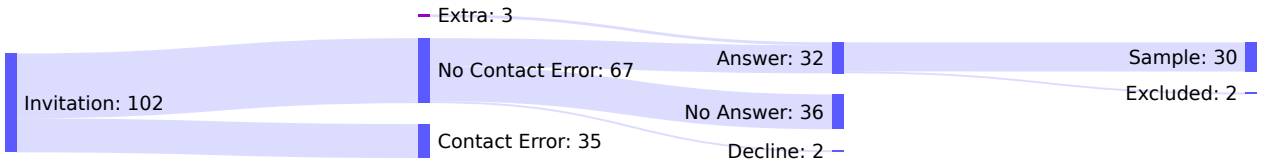

Figure 1: A depiction of our process flowing from sending invitations to the left to obtaining the sample to the right.

2 METHOD

The purpose of our survey was to establish current practices in DSL's life cycle by sending a questionnaire [13] to DSL authors. The fundamental method for our investigation was to first identify a number of relevant DSLs, then send the authors of these DSLs an invitation to the questionnaire regarding their specific DSL, and, finally, analyse the responses. We sought to obtain a representative sample of DSLs of interest to academia and industry by surveying DSLs from curated DSL collections. We sent the first round of invitations during the Summer of 2021, and we sent a second round of re-invitations during the Winter of 2021 to authors we had been unable to contact. We postpone discussions of methodological threats to validity to Section 5.

We asked all invitees to answer the questionnaire regarding a specific DSL project through email or a similar contact method. Figure 1 shows how the original 102 invitations resulted in a sample of 30. We received an error message from 35 of our invitations, meaning that 67 DSL authors received an invitation at most. We allowed participants to make submissions about other DSL projects they found relevant. We received and included three such submissions, which we label as *extra*. We obtain an effective response rate of at least $29/67 = 43\%$ when not counting *extra* answers.

2.1 Invitations

We used existing curated collections of DSLs as our pool of invitation candidates. We chose the following collections to ensure that the investigated DSLs were of interest to academia and industry:

- The paper, *Domain-Specific Languages: An Annotated Bibliography* [39]
- The workshop series proceedings of *Real World Domain Specific Languages* [3, 4, 6, 7]
- The conference proceedings of *Domain-Specific Languages '99* [1]
- The workshop proceedings of *Functional Programming Concepts in Domain-Specific Languages* [2]
- The workshop proceedings of *Domain-Specific Languages Design and Implementation* [38]
- The collection of financial DSLs, *dslfin.org* [10]
- The collection of DSLs, *Wikipedia categories* [5]
- The collection of robotic DSLs, *Robotics DSL Zoo* [9, 30]

We had two exclusion criteria for filtering out artefacts appearing in these collections. We removed an artefact if it a) was not a DSL or if b) the authors did not intend for their DSL to be used. Criterion *a* was primarily important since these collections contain many methods and tools not relevant to the survey. When making this decision, we considered 1) if it was stated that the artefact was a DSL, 2) if the artefact had textual or visual language constructs tailored to a specified domain, and 3) if the application domain of the artefact was narrow. We primarily removed non-language artefacts, but we removed a couple of artefacts since we analysed them to have too many general-purpose characteristics to include. One example of such a language is the programming language Q# [36], designed for expressing quantum algorithms seamlessly alongside classical computations. While the domain of Q# is quantum algorithms, we find that its domain application domain of quantum computations and classical computations is an extension of many general-purpose languages. For criterion *b*, we were lenient in deciding whether a DSL was not intended to be used by only excluding DSLs where the authors explicitly stated they were not. This leniency is also apparent from the received answers, where two participants explicitly stated that the purpose of the DSL was not intended for use. We excluded these submissions from the final sample.

2.2 Questions and Data Interpretation

Besides factual questions, the questionnaire consisted of a) multiple-choice questions allowing multiple answers and free-text answers and b) free-text questions. Only a few factual questions were required to be answered by participants. All of these questions were easily answerable by all participants to not create a barrier to participation.

Free-text answers were analysed and grouped into representative categories using open coding. We either created a new category or used a predefined choice when appearing in multiple-choice, multiple-answers questions. To be transparent about this process, we emphasise new categories in figures, and for each category, we visualise the fraction we interpreted to belong there. We used only new categories when interpreting answers to free-text questions since there were no predefined choices for these questions. We assigned a single free-text answer to several categories when we found it to describe a multitude of subjects. In the presentation, we show representative cases for each category. These answers have been anonymised and edited for presentation while preserving their original meaning. We excluded a couple of free-text answers from interpretations since they contained apparent mistakes. When we encountered such answers, we ensured that the remaining answers of the submission were coherent.

3 RESULTS

The survey questionnaire focused on five topics: 1) DSL characteristics, 2) evaluation and user involvement in design, 3) launch, 4)

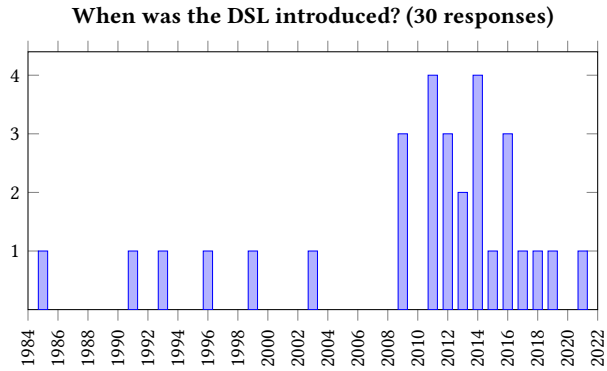

Figure 2: Data plot of when the surveyed DSLs where introduced. The mean age of the DSLs is 11 (introduced in 2010).

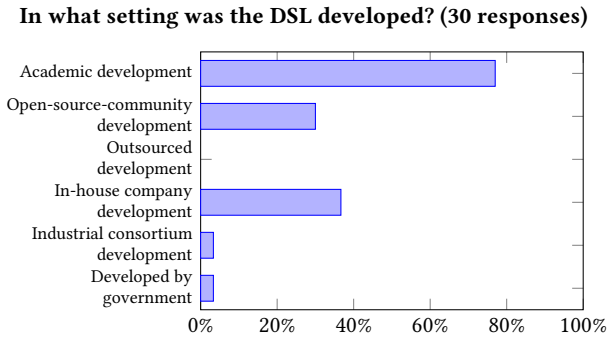

Figure 3: Data plot of development setting of surveyed DSLs.

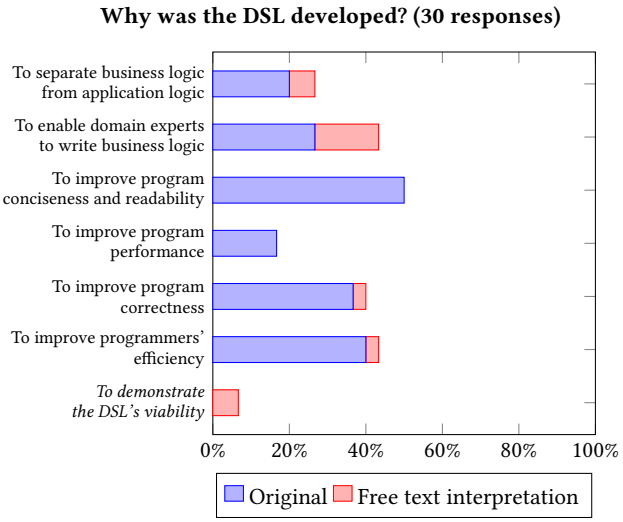

Figure 4: Data plot of reasons for developing the DSL.

evolution, and 5) end of life. We treat each topic in its own section by first giving an executive summary, then presenting the results, and finally discussing our findings. We cross-examine data from different topics whenever relevant. We report answers to multiple-choice questions in percentages and answers to free-text questions in the number of respondents. Findings and claims are numbered using circled numbers Ⓧ which permit later referencing.

3.1 DSL Characteristics

We present the characteristics of the 30 DSLs in our sample, i.e., the age of the DSLs, their development setting, and the reason for creating them[1]. The characterisation of the sample forms a basis for the kind of DSLs our subsequent findings are generalisable to. The surveyed DSLs are predominantly developed in an academic setting and are on average 11 years old, with the majority being younger.

Results. The DSLs in the sample are between 0 and 36 years old (introduced between 1985 and 2021), with an average age of 11 years. While this is a broad range, 80% of the DSLs are younger than 12 years (Figure 2). Of the surveyed DSLs, 77% were developed within an academic setting (Figure 3). Still, of these DSLs developed in an academic setting, 52% were also partly developed within some other setting. Of the surveyed DSLs, 30% were developed in an open-source community, 38% were developed in an industrial setting[2], and 19% were developed in both an academic and industrial setting. The sample represents all of our suggested reasons for developing a DSL (Figure 4). While 50% of DSLs sought "to improve program conciseness and readability", only 3% had this as their only reason. Comparatively, 55% of DSLs were created for more technical reasons such as "to separate business logic from application logic" or "to improve program performance".

Discussion. We confirmed our anticipation of having a majority of academically developed DSLs since we primarily invited DSLs from academic collections (see Section 2). We find that while it is often mentioned that DSLs allow for a higher-level description of a domain, this is rarely the only reason for creating a DSL. Authors often seek other effects allowed by the higher-level description, such as improving program correctness or programming efficiency. In all, ① we find that the sampled DSLs primarily represent academically developed DSLs with some industrial uses that were developed for many different reasons and have an age between 5 and 12 years.

3.2 Evaluation and User Involvement

Expert evaluation is the most used evaluative method during DSL design, but different user-centred evaluation methods are also commonly used. We find no established practice for the level of user involvement during the DSL design, and we do not find any correlation between the degree of user involvement and a DSL's introduction year or the level of users' programming experience.

Results. ② Expert evaluation is the predominant method for evaluating a DSL during design and development, and it is used by 57% of DSLs (Figure 5). Of these, 41% use it as their only evaluative method.

[1] We are also interested in the type of users presented in Section 3.2.
[2] This category consists of in-house development, industrial consortium development, and governmental development.

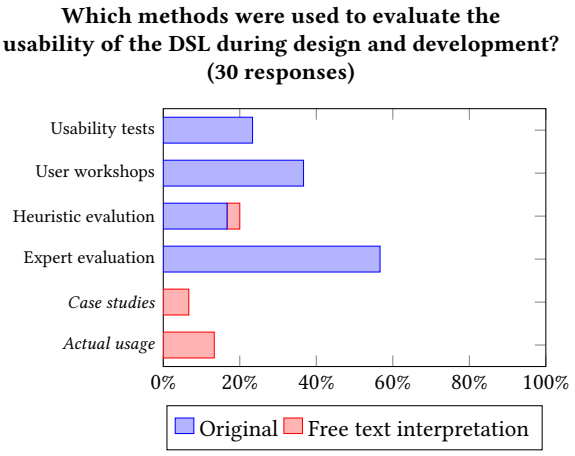

Figure 5: Data plot of broad categories of evaluative methods. We have shown free-text answers mentioning different kind of case-studies as their own category, although they could also be considered as a form of expert or heuristic evaluation depending on how they were conducted.

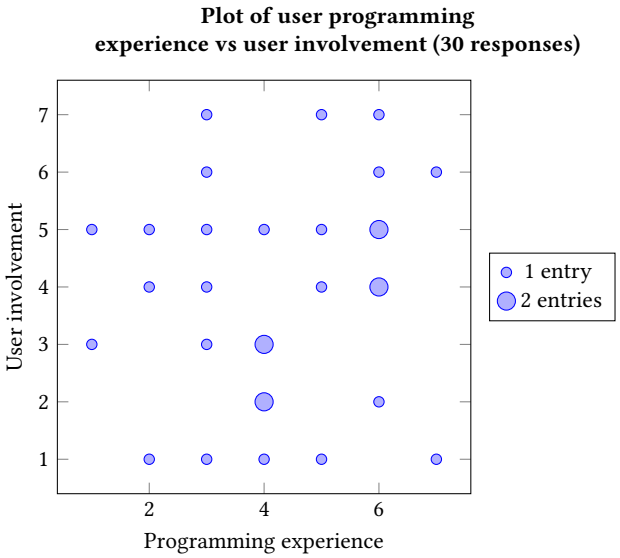

Figure 6: A data plot of users' programming experience vs. the level of user involvement in DSL design. The dimensions range from none (score 1) to expertise/involvment in all decisions (score 7).

Different human-centred evaluation methods[3] are used by 53% of DSLs. Finally, ③ heuristic evaluation is used only by 20% of DSLs, even though the method does not require any user involvement.

We find that DSLs are created for target users who have vastly different levels of programming experiences ranging from none (score 1) to expert (score 7) (Figure 6). While there are DSLs that target users with no programming experience, it is more common for DSLs to target users that have at least some programming experience. We also find that there is a wide range of involvement of users in the design process ranging from none (score 1) to involvement in every decision (score 7) (Figure 6). ④ Users are primarily involved in designing the DSL by assisting in requirement elicitation and by giving feedback during user tests, workshops, or from actual usage (Table 1). For requirement elicitation, users are involved in outlining the domain of the DSL, either by telling what kind of problems they usually face, by more actively designing scenarios, or by being involved in designing the DSL itself. The dominating methods for obtaining feedback on a DSL design are testing with users, conducting workshops, and actual language usage. Six respondents mentioned that these techniques are used iteratively through several increments of the DSL. Different kinds of language demonstrations are also used to obtain feedback from users through iterative interactions. It is common for DSLs to be designed by their users, either by having the designer being a user themself or by having a user within the design team.

Discussion. ⑤ We conjecture that the evaluative technique of expert evaluation has a significant impact on the usability of created DSLs due to its widespread usage. Also, we find that many DSL authors are interested in having users evaluate their DSL during design and development seen by the usage of human-centred methods.

For the programming experience of users, we expected most DSLs to be designed for users with some programming experience since we expected most DSLs to be developed for a domain where programming activities have become necessary. We find it more surprising that 17% of DSLs are created for users with little to no programming experience.

We find that there are large differences in how users are involved in the design project and the level of involvement ⑥. When the users of the DSL are part of the design team, we find it safe to presume that these in-team users are involved to some extent with most aspects of the language design and thereby impact the design significantly. We hypothesised that there could be a correlation between the level of user involvement in the design process and the programming experience of users. It is reasonable to involve users in the design of the DSL because they, as experts, are qualified to give informed feedback. On the contrary, it is also reasonable to involve users with little experience to ensure they understand the designed concepts and find them adequate. However, ⑦ the Kendall rank correlation coefficient of 0.09 between programming experience and user involvement shows no correlation with a p-value of 0.52. We made similar correlation tests between the year of DSL introduction and programming experience or user involvement. Both of these tests found no correlation with p-values of respectively 0.33 and 0.84.

[3]This category consists of usability tests and user workshops.

Table 1: How were users involved in the language design? (30 responses)

Requirement elicitation (7 answers)	• We asked the users to specify mathematically the problems they usually face. • Use case driven approach, elaborating on domain level specification of specific use cases.
Feedback (8 answers)	• Via conferences and mailing lists, where they could express their opinions. • Workshops for the initial and later incremental versions. • Feedback from a small in-house group. • Continuous user testing.
In team (4 answers)	• Prospective users were in the design team. • A small set of expert users were involved from the beginning in all aspects.
Designer is the user (3 answers)	• I am a user myself and was part of the creation and design of the language.
Actual/early usage (2 answers)	• Make them use the initially released versions; collect feedback to improve subsequent releases. • Users used the initially released version; they gave feedback for later releases.
Other (5 answers)	• Feature-Oriented Domain Analysis. • Through case studies.
No involvement (2 answers)	• The language is primarily machine-to-machine oriented.

3.3 Launching

DSL authors use a wide range of techniques to attract a user base when launching a DSL. Such techniques often affect the success of a DSL, and it is especially important to demonstrate the merits of using the language to attract users.

Results. The language ecosystem (development environment, training material, and language documentation) is used by 73% of DSLs to encourage usage. Smoothing the transitioning to using the DSL by either being backwards compatible or assisting in the transitioning is part of 30% of DSLs' launch strategy. Another approach used by 13% of DSLs is to require usage through company policy. Seven respondents opted to provide free-text answers focusing on promotion and social initiatives instead of the more technical approach suggested by the proposed answers.

Of 23 respondents, 9 answered that the initiatives for encouraging DSL usage affected their DSL's success (Table 2). Two topics reoccur in several responses. First, the initiatives are decisive for the success of the DSLs. Second, the assistance in transitioning from old solutions sometimes mitigates a steep learning curve. Again, seven respondents replied that the success of the DSL is primarily affected by its merits and the demonstration of these.

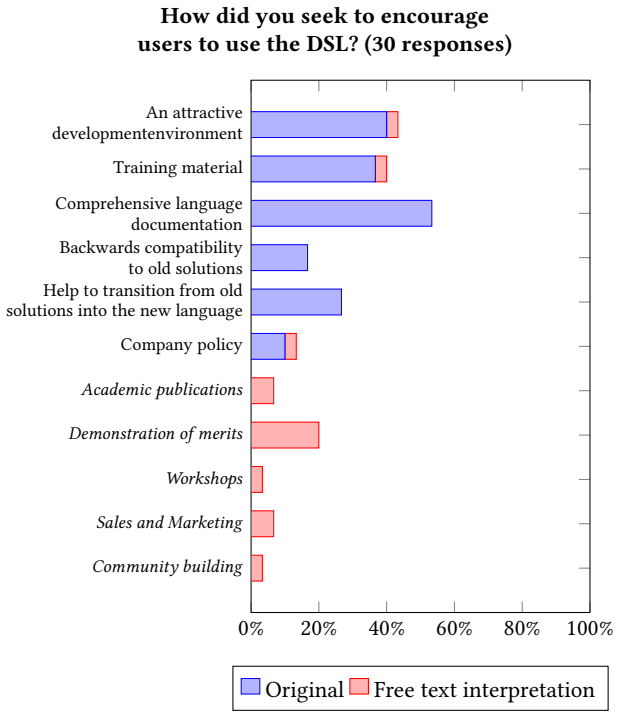

Figure 7: Data plot of techniques for encouraging DSL usage.

Discussion. While our proposed reasons for user encouragement primarily focus on implementation efforts, free-text answers focus more on demonstrating the merits of the DSL, social initiatives, and promotion. We would expect more participants to choose even more of such options if we had originally proposed them. We find that respondents emphasise that the merits of a DSL should present an improvement in the quality of life of its users compared to its alternatives. One respondent mentions that, for them, the improvement is so significant that the alternative of not using the DSL is unattractive. We interpret these answers relating to social initiatives and promotion to indicate that they see these aspects as necessary. Therefore we find that (8) it is essential to demonstrate the primary parts of the DSL to users, namely its merits. This demonstration can occur through different channels such as academic publications, sales and marketing, or community building.

3.4 Evolution

Almost all DSLs evolve after their launch, which is experienced by users as, among others, new language constructs and improvements in language implementation. How the evolution is performed is important for the success of a DSL, whether it is improving the existing language implementation, finding new application domains, or improving the development environment.

Results. (9) Evolution is a pervasive phenomenon for DSLs, with 86% of respondents of the entire sample reporting at least one cause of evolution. Almost all suggested evolution factors are equally common (Figure 8). Of respondents, 60% are affected by factors

Table 2: Do you think these efforts affected the success of the DSL? Why? (23 responses)

Affirmative (9 answers)	• Yes, but some users prefer to stick to the GUI. • Yes. I don't think people would have adopted it if we hadn't actively promoted and sold it. • Yes, the transition from earlier languages was effective.
Language merits important (7 answers)	• It succeeded really as a step-change in what could be done in terms of quality: there are many other DSLs in this general domain, but today only a few are meaningfully used. These present such opposite extremes that there is no real decision point in a single project as to which to use.
Other (7 answers)	• Users had by policy to use the DSL. • I would not characterise the DSL as very successful, but it does allow the users to quickly extend the existing system with non-standard products.

What factors have contributed to evolving the DSL after its launch? (29 answers)

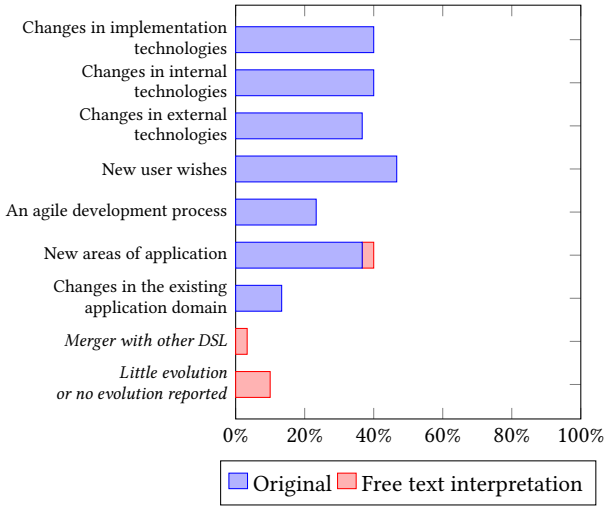

Figure 8: Data plot of evolution causes.

they are somewhat in control over, such as accommodating user wishes, having an agile development process, seeking new areas of application, and making changes to technologies internal to the maintainer. Still, 63% are affected by factors outside of their control, such as changes to the application domain, implementation technologies, and external technologies.

From the users' perspective, the most common form of DSL evolution is the addition of new language constructs or syntactic sugar (Figure 9). It is comparatively rare that major changes are made to the existing syntax, which happens to 23% of DSLs. (10) 17% of DSL creators have found it necessary to introduce breaking updates. A

Which changes have been made to the DSL from a user perspective? (29 answers)

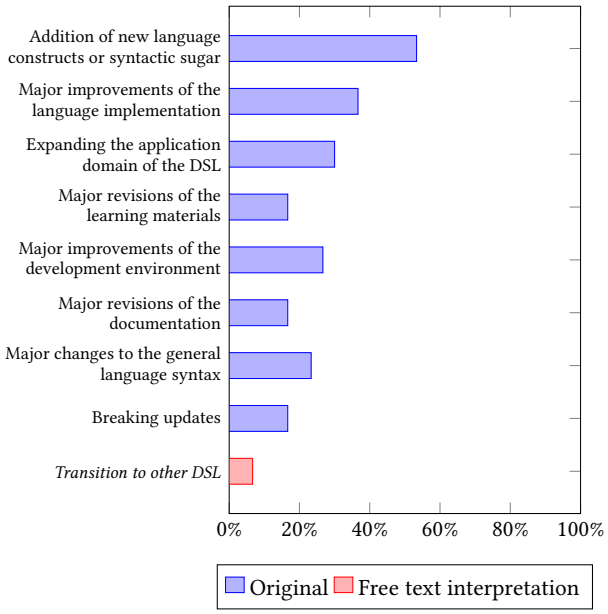

Figure 9: Data plot of DSL changes from the user perspective. Judging whether is major was left to the discretion of participants.

major evolution of the ecosystem of DSLs also occurs but is rarer.

(11) Of 21 respondents, 13 answered that evolution of a DSL is important, if not vital, to the success of DSLs (Table 3). As we have already shown, the evolution may take many forms, such as improving the quality of the DSL, accommodating user wishes, and making it more accessible and applicable. These improvements are important since they may introduce benefits that did not originally exist and help keep the DSL relevant. One respondent who mentions that evolution did not significantly affect their DSL's success even states that "I suspect those changes did not affect the language's success, except that if it had never made any progress, then it would have faded away."

Discussion. We identify that a DSL's constituents have different evolutionary characteristics. The questionnaire indicates that often the DSL (language constructs, implementation, and domain) itself is susceptible to evolution caused by new user wishes and domain evolution. Comparatively, the ecosystem of the DSL is less susceptible to major evolution. We hypothesise that this difference is due to the developers having more control over the ecosystem of the DSL than the usage of the DSL.

We expected that most DSL authors would rarely introduce breaking updates to their language. Surprisingly, we found that breaking updates are not that uncommon. We hypothesise that

Table 3: Do you think these efforts affected the success of the DSL? Why? (21 answers)

Vital (1 answer)	• Continuous improvement is necessary to keep the language alive. Later this was not possible without disappointing users.
Affirmative (12 answers)	• Yes, we designed modular, reusable components, which became a major benefit of adoption. • Yes, we improved performance, reliability, and usability.
Not significantly (1 answer)	• I suspect those changes did not affect the language's success, except that if it had never made any progress, then it would have faded away.
Inconclusive (2 answers)	• Unclear if they helped with the success.
Language merits important (1 answer)	• Success mainly due to stability of the DSL and reuse of example DSL specifications (library).

Life span of DSLs (28 answers)

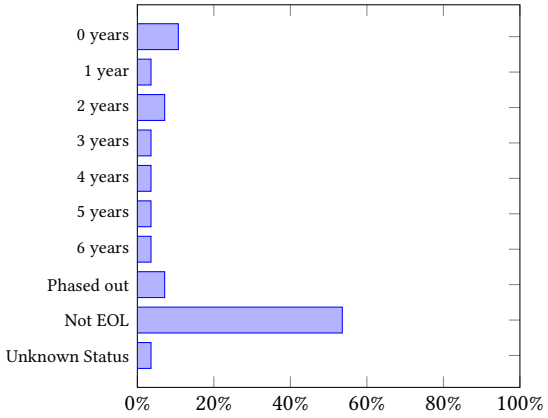

Figure 10: The life span of the surveyed DSLs. The findings stem from the answers to two different questions where some participants stated their DSL where *phased out* without mentioning a specific year.

breaking updates occur when DSL author has pragmatically estimated that the cost of migrating DSL programs is sufficiently low. We find this form of pragmatism to be a characteristic of DSLs as opposed to general-purpose languages where non-breaking updates, for the most part, are unthinkable.

3.5 End of Life

In investigating the end of life (EOL), we recognise that a) it can be complicated to say when a DSL is phased out and b) it is impossible to foresee when, why, and if a DSL will be phased out. Therefore, we asked participants to answer questions to their best ability even if their DSL was not at the EOL. We emphasise that these answers are a mixture of actual experiences and estimates.

Results. Many DSLs are long term software projects that remain in use a decade after their introduction (Figure 10). Of the surveyed DSL, 53% of authors did not consider their DSL project to have reached its EOL. These DSLs have an average age of 12 years, and several of them estimate that they will remain in use for 5, 10, 15, or more years. (12) DSLs that had reached their EOL were commonly retired either due to changes in their domain, better tooling, or adoption by other languages (Figure 4). DSLs that were not phased out yet answered similarly for the expected reasons of their retirement with a focus on new languages or tooling replacing the DSL.

Finally, we asked survey participants whether there were any efforts that the DSL authors would have taken with the power of hindsight (Table 5). While this question does not investigate features of the surveyed DSL as such, we found it relevant to examine whether DSL creators had any prevalent lessons learned. Besides the authors who answered they would have made no efforts looking back, the answers highlight five different aspects. First, it is inevitable to have such efforts that they would have made. Second, creators should remember activities for attracting users, either through promotion or internal usage. Third, there are different ways of improving the development process. Such improvements are considered both from a process perspective and an architecture perspective. Fourth, the application domain can be difficult to work in, and creators can consider switching to other application domains. Fifth, case studies during development could have improved the language, but creators were unable to do so due to case studies

Table 4: What was, or will be, the primary reason for the DSL to have been phased out? (Best estimate if in the future) (21 answers)

Ecosystem changes (2 answers)	• Dependent on obsolete infrastructure. • Software base and applications moved on, and we lacked resources to keep the DSL updated.
Replaced by other language or tooling. (7 answers)	• Replaced by the SQL standard, making a standalone DSL somewhat redundant. • Replaced by new tooling • Better database design and centralised solutions mooted the problem being solved.
Domain changes (3 answers)	• Domain itself phased out. The language was reborn in new domains. • Important changes in the domain.
Not EOL (2 answers)	• This DSL is not EOL.
Lack of users (2 answers)	• Lack of continued commercial support, a need to rewrite the systems based on it, and lack of available expertise if usage declines.
Other (2 answers)	• Development team has moved to other projects.

Table 5: Looking back, are there any efforts that you would have taken during any of the previous phases? Why did you not take them at the time? (23 answers)

Usage and promotion (4 answers)	• We considered but ultimately chose not to follow to exchange hardcoded specifications for their DSL counterparts. However, the language and implementation are not sufficiently developed to support all specifications. • Promoting its use. This was not done because of the DSL's academic character and limited resources.
Usability (1 answer)	• Make it more user friendly: make the compilation more robust to failure for various corner cases; improve error reporting of syntactic/semantic errors.
Improve development (3 answers)	• Application context, domain application, and language evolved separately. It is possible that we could have unified them from the start with a bottom-up approach with nothing working end-to-end along the way. • Waited longer before committing to a DSL; rather, go for an opinionated library in a suitably flexible host language first. • We should have looked sooner for a fundamentally simpler architecture for the underlying system instead of assuming that it was fundamental.
Application domain (2 answers)	• The bottleneck for use was never the language itself but state of the art in the domain systems it targeted. • We could have focused more on a broader domain earlier, and we could have pushed on off the shelf reusable components earlier. We could have focused more on fault tolerance, including better support for lower performance applications.
Use case studies (2 answers)	• Developing larger examples in the language would have helped. Doing that while developing the language and preparing the paper as part of a large team was challenging at the time. • Consider more complex use cases during the DSL specification; impossible due to lack of human resources.
Inevitable (2 answers)	• In a language that develops incrementally, at certain points, one is likely to wish one had done some things differently, but they can't be redone on account of maintaining backward compatibility. • One always has a few regrets: features hastily added to the language early on, which now seem redundant or awkward, for example, or implementation decisions that made the compiler code-base harder to maintain decades later on. But basically, no, I wouldn't do much of it differently if offered the chance.
Negative (5 answers)	• No
Other (1 answer)	• While interesting, DSL development is not my primary research field.

being a labour intensive endeavour.

Discussion. We find that while a DSL serves as an alternative to using a GPL, it may be so successful that it is adopted by GPLs or other tools. This adoption is explicitly mentioned by three respondents, and it is also mentioned implicitly by three others since they expect their DSL to become obsolete when more mainstream tools or languages adopt the functionality the DSL provides. That is, there is a dual movement of ideas between DSLs and in GPLs or tools in that DSLs are created due to inefficiency in GPLs or tools, which in turn may evolve seeing the utility demonstrated by the DSL.

While we identified ⑧ promotion to be an important aspect of launching a DSL, the retrospective answers indicate that promotion may be a somewhat neglected activity. We also consider what DSL authors did not mention as problems retrospectively. While we found that evolution is important for the success of many DSLs ⑪, only three answers relate directly handling evolution differently, not counting those that discuss looking to expand the DSLs' domain. Likewise, while four answers consider usability and use cases of the DSL, there are no explicit regrets as to how users were involved in the design process. ⑬ These findings indicate that current ways of having user involvement in the design process are not causing dissatisfaction among DSL creators.

4 RECOMMENDATIONS

Based on the insights gained in our survey, we have six recommendations for DSL practitioners, educators, and the field in general. In doing so, we seek a balance between, on the one hand, recognising the value of the established practices mentioned by DSL authors and, on the other hand, challenging these practices.

R1: Explore practices for expert evaluation of DSLs. We found that expert evaluation is a widespread evaluation technique used by 57% of respondents ②. From this result, we conjectured that the method of expert evaluation has a significant impact on the design of many DSLs ⑤. However, expert evaluation is susceptible to the profile of the evaluating expert and how the expert evaluates the language. While we also found that 22% of respondents use heuristics evaluation, there still is a noticeable gap in unexplored practices. Therefore, we recommend further investigations into practices on expert evaluation of DSLs with two purposes: First, academia should explore how expert evaluation is performed, by whom, using which artefacts, and to what effect. One should compare the effectiveness of using expert evaluation against comparable techniques. Second, due to the widespread use of the techniques, we find a need for academia and industry to establish guidelines for practitioners jointly. We hypothesise that the guidelines should

be low-cost and easily approachable if they are to be followed.

R2: Guidelines for the level and kind of user involvement. We find that DSL creators have vastly different approaches to the kind and level of user involvement during DSL design (6). The user involvement ranges from none at all, to requirement elicitation, to use-case design, to language evaluation, to users being part of the design team. Although we found no indication of dissatisfaction regarding user involvement (13), we still find a need for guidelines for how and when users should be involved in a specific design project. We hypothesised that there could be a correlation between the level of user involvement and the programming experience of users but found none (7). Still, we believe that the prescriptions should be based on the design context, such as who is the target user, how available are users, what kind of tasks are they to perform, and how complex is the DSL.

R3: Consider the purpose of heuristic design principles. We found that heuristic evaluation is used only by 22% of survey participants (3). Still, heuristic design principles are valuable knowledge since they are the expressed experience of experts within the field. We hypothesise that the utility of heuristic design principles could increase if they allow for more lightweight evaluation activities or can be applied generatively in the design process. Therefore, we recommend that the creators of heuristic design guidelines consider how and when these guidelines should be used in DSL design.

R4: Use the flexibility allowed by the project. We find that DSL authors report many different ways of being flexible or pragmatic in their approach to DSL design. To mention three examples: First, authors found it necessary to introduce breaking updates to their languages. One respondent reported that this was necessary to accommodate a shift in research focus to a different underlying technology. Second, authors found new application domains for their DSL project. One respondent reported that their DSL became more successful by widening the range of applicable application domains. Third, authors report major improvements made to the DSL and its launch. These improvements come in many forms, such as performance, usability, and modularity. From these insights, we recommend that DSL practitioners should use the flexibility allowed by their project as opposed to being dogmatic in their development approach.

R5: Consider evolution as an intrinsic part of DSL creation. We found that evolution is a part of most DSLs' life cycles and that this evolution is important for the success of many DSLs (9) (11). We have already mentioned some examples of the importance of evolution in recommendation *R4*. While this finding may not be surprising for many practitioners and scholars within the field, we find it essential to substantiate this claim empirically. From this finding, we have a three-fold recommendation. First, creators of DSLs should consider evolution as an aspect of DSL creation. Second, tool creators should continue developing tools with dedicated support for managing the evolution of language specifications and artefacts. Third, educators, methods, and textbooks on DSL design should treat evolution with utmost importance.

R6: Consider EOL through adoption a success criterion. We found that DSL authors report that their DSL reached (or will reach) EOL due to the DSL's functionality being adopted by tools or other languages (12). One example is a language that has become obsolete since its functionality was incorporated into a mainstream and widely used standard. While the EOL of a DSL through being absorbed into another language or tool is commonly viewed as defeat, we recommend declaring this as one potential success. Furthermore, we recommend using the means of language engineering for the DSL to influence standard concepts and constructs used to represent the domain.

5 THREATS TO VALIDITY

We have mitigated threats to validity originating from our sampling method and our way of conducting the questionnaire.

Internal Validity relates to the degree to which we can trust the findings within our survey. There is an internal validity risk of not measuring the intended phenomenon when conducting a questionnaire. We mitigated this risk of measuring something unintended in four ways. First, we presented and discussed both the questionnaire's purpose and questions with a colleague. Second, we conducted a blind pilot run of the study with another colleague and subsequently discussed their understanding of the asked questions. Third, for all answers that did not solely establish a fact, we allowed participants to submit free-text answers allowing them to answer a question differently than we had intended. Fourth, in our presentation, we are transparent in what interpretation we have made.

Another risk to internal validity is that there is insufficient data for the claims made in our findings. Our primary mitigation of this risk was to obtain a sample of sufficient size so as to be less susceptible to noise. Also, while we do not find any correlation, we find the sample size to be adequate for our chosen method. We found that including *extra* submissions did not change our statistical findings but did provide valuable examples.

We considered excluding DSLs introduced before 1990 to avoid too diverse subject DSLs since this might threaten internal validity. However, old DSLs should be included to increase internal validity since an exclusion criterion comes with the following three methodological problems. First, the survey seeks to investigate the entire life cycle of DSLs, meaning that older DSLs are relevant for the survey. Second, using the exclusion criterion, we would have presupposed the lifetime and life cycle of DSLs. Third, if an old DSL is still in use, then it is at least as interesting as a more modern one when examining the current state of DSLs' life cycle.

External Validity relates to the degree to which we should believe our findings to be transferable. From this perspective, the most severe threat is the application domain of the surveyed domain. We recognise that the chosen collections of DSLs to sample from are biased towards DSLs within finance and robotics domains. Therefore, our survey has the highest degree of external validity when generalising to other DSLs within these and similar domains.

Still, we have mitigated the influence of the bias on our findings by avoiding questions directly influenced by the domain, such as tooling, application context, general kinds of domain tasks, monetary costs, and underlying programming paradigms.

Another potential bias in the sample is that the survey sample does not represent the actual population of DSLs. We recognise that our sampling method is biased towards receiving answers from newer publications for a multitude of reasons, e.g., authors changing email addresses, authors retiring, DSL projects retiring, companies dissolving, or newer authors being more excited about their work being noticed. Therefore, we present the age to characterise the DSL sample to make the bias transparent. We had mitigation measures seeking a sample as representative as possible. First, we have sought multiple channels of contacting all invited authors. Second, to avoid participants getting stuck on non-applicable questions or questions participants could not answer, we allowed skipping questions. We deemed that older DSLs had a higher risk of encountering these kinds of problems. Third, to accommodate participants who did not want to submit data on the third-party platform, we allowed participants to email their answers.

6 RELATED WORK

Several studies seek to investigate the current state of different DSL topics through meta-studies of the field. These studies, which we describe below, use three different sources to obtain information. First, *zoo analysis* considers software itself as the primary source of information. Second, *publication analysis* considers publications as the primary source of information. Third, *questionnaire analysis* uses self-reports from questionnaire recipients as the primary source of information. Both zoo analysis and publication analysis come with strengths and weaknesses compared to the questionnaire approach. Zoo analysis does not rely on interpreting DSL authors' possibly biased reporting, but they cannot answer development-oriented research questions. Publication analysis may answer development-oriented research questions but relies on authors' prior reports on areas of interest and can only answer questions to the degree that report uniformity and granularity allows.

Using zoo analysis, Dragule et al. survey DSLs for robotic missions [15]. They identify and categorise 30 robotic mission programming environments and present their design space through a feature model of the environments. Similarly, Kapre and Bayliss [22] survey 9 DSLs used for high-performance FPGA computing. Schauss et al. create a chrestomathy of DSL implementations to teach implementation techniques [34]. They subsequently conduct a pure zoo analysis of their implementation to identify implementation variations of interest. For this purpose, several DSL zoos are open for future zoo analysis [8, 9].

Using publication analysis, Deursen et al. [39], Marnik et al. [28], and Oliveira et al. [31] all made early investigations into DSL development and implementation methodologies by reviewing selected publications. More recently, Nascimento et al. [29] and Kosar et al. [24] have conducted systematic mapping studies to investigate research within the field. Of relevance to our study, Kosar et al. find that much research proposes new techniques supporting different development phases with an overwhelming focus on design, implementation, and domain analysis, with very few considering maintenance and validation. Iung et al. [21] conducted a similar systematic mapping study for tools being used by DSL creators. On the same topic, Erdweg et al. [16] evaluate and compares language workbenches.

Systematic literature reviews are also used as a form of publication analysis. Like our paper, Poltronieri et al. investigate how DSL authors evaluate their DSLs' [33] and create a taxonomy for evaluation on this basis. In an updated review [32], they find usability evaluation to be the most often used evaluation technique. However, they also find that even after applying a quality assessment filter only, 13 out of 21 describe their used technique. This finding points to a methodological difficulty in examining some topics through publication analysis. As such, our method can be seen as a different angle of attack with its own threats to validity.

Questionnaire analyses have been used to explore the broader topic of model-based engineering. For example, Broy et al. [14] investigate the benefit of using model-based practices within the car industry, Badreddin et al. [11] investigate trends in software practitioners' use of model practices, and Liebel et al. [26] investigate students' perception of modelling tools and UML.

7 CONCLUSION

In this paper, we have presented a survey to establish current practices in managing the life cycle of DSLs through a questionnaire. The 30 answers from the authors of DSLs provide us with several findings relating to the DSL management phases of the design and development, launch, evolution, and end of life. Among others, we find that a) there is no established practice as to the level of user involvement during development, b) DSL authors find demonstrating the merits of a DSL is important during launch, c) handling evolution correctly is important for the success of a DSL, and d) that it is common for DSLs to be replaced by other tooling or languages. Based on our findings, we have presented six recommendations relating to different phases of a DSL's life cycle. Among others, we recommend a) further explorations of practices for expert evaluation, b) that DSL practitioners are flexible in their approach to DSL development, and c) that DSL evolution as a topic is treated with utmost importance both in education, industry, and academia. For future work, we propose that an open research database is created where DSL creators register and update information on their DSL. Such a database would provide the community with high-quality information and allow more process-oriented research but require a method for deciding what the database should contain.

8 ACKNOWLEDGEMENT

We thank Innovation Fund Denmark (7076-00029B) for funding, all questionnaire recipients for their time and contributions, and Peter Sestoft for invisioning and supervising the work.

REFERENCES

[1] 1999. DSL '99: Proceedings of the 2nd Conference on Domain-Specific Languages. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/331960

[2] 2013. FPCDSL '13: Proceedings of the 1st Annual Workshop on Functional Programming Concepts in Domain-Specific Languages. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2505351

[3] 2016. RWDSL '16: Proceedings of the 1st International Workshop on Real World Domain Specific Languages. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2889420

[4] 2017. RWDSL17: Proceedings of the 2nd International Workshop on Real World Domain Specific Languages. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3039895

[5] 2018. Category:Domain-specific programming languages. https://en.wikipedia.org/w/index.php?title=Category:Domain-specific_programming_languages&oldid=858848463 accessed 1 May 2021.

[6] 2018. RWDSL2018: Proceedings of the Real World Domain Specific Languages Workshop 2018. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3183895

[7] 2019. RWDSL '19: Proceedings of the 4th ACM International Workshop on Real World Domain Specific Languages. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3300111

[8] 2020. DevBoost/EMFText-Zoo. https://github.com/DevBoost/EMFText-Zoo original-date: 2012-08-07T07:22:54Z.

[9] 2020. Index – Robotics DSL Zoo. https://corlab.github.io/dslzoo/all.html accessed 1 May 2021.

[10] 2021. Financial Domain-Specific Language Listing and Resources. http://www.dslfin.org/resources.html accessed 1 May 2021.

[11] Omar Badreddin, Rahad Khandoker, Andrew Forward, Omar Masmali, and Timothy C. Lethbridge. 2018. A Decade of Software Design and Modeling: A Survey to Uncover Trends of the Practice. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, Copenhagen Denmark, 245–255. https://doi.org/10.1145/3239372.3239389

[12] Ankica Barišić, João Cambeiro, Vasco Amaral, Miguel Goulão, and Tarquínio Mota. 2018. Leveraging teenagers feedback in the development of a domain-specific language: the case of programming low-cost robots. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. ACM, Pau France, 1221–1229. https://doi.org/10.1145/3167132.3167264

[13] Holger Borum. 2022. Artefact: Survey of Established Practices in the Life Cycle of Domain-Specific Languages. https://github.com/hborum/models-22-survey original-date: 2022-07-22T06:26:06Z.

[14] Manfred Broy, Sascha Kirstan, Helmut Krcmar, and Bernhard Schätz. 2012. What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry? https://doi.org/10.4018/978-1-61350-438-3.ch013 ISBN: 9781613504383 Library Catalog: www.igi-global.com Pages: 343-369 Publisher: IGI Global.

[15] Swaib Dragule, Sergio García Gonzalo, Thorsten Berger, and Patrizio Pelliccione. 2021. Languages for Specifying Missions of Robotic Applications. In *Software Engineering for Robotics*, Ana Cavalcanti, Brijesh Dongol, Rob Hierons, Jon Timmis, and Jim Woodcock (Eds.). Springer International Publishing, Cham, 377–411. https://doi.org/10.1007/978-3-030-66494-7_12

[16] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (Dec. 2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007

[17] Marco Frigerio, Jonas Buchli, and Darwin G. Caldwell. 2013. A Domain Specific Language for kinematic models and fast implementations of robot dynamics algorithms. *arXiv:1301.7190 [cs]* (Jan. 2013). http://arxiv.org/abs/1301.7190 arXiv: 1301.7190.

[18] Ruediger Gad. 2017. Evolution of a Stream Transformation DSL. In *Proceedings of the 2nd International Workshop on Real World Domain Specific Languages - RWDSL17*. ACM Press, Austin, TX, USA, 1–10. https://doi.org/10.1145/3039895.3039897

[19] Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2009. Domain-Specific Languages in Practice: A User Study on the Success Factors. In *Model Driven Engineering Languages and Systems (Lecture Notes in Computer Science)*, Andy Schürr and Bran Selic (Eds.). Springer, Berlin, Heidelberg, 423–437. https://doi.org/10.1007/978-3-642-04425-0_33

[20] Nico Hochgeschwender, Sven Schneider, Holger Voos, and Gerhard K Kraetzschmar. 2014. Declarative Specification of Robot Perception Architectures. (2014), 12.

[21] Aníbal Iung, João Carbonell, Luciano Marchezan, Elder Rodrigues, Maicon Bernardino, Fabio Paulo Basso, and Bruno Medeiros. 2020. Systematic mapping study on domain-specific language development tools. *Empirical Software Engineering* 25, 5 (Sept. 2020), 4205–4249. https://doi.org/10.1007/s10664-020-09872-1

[22] Nachiket Kapre and Samuel Bayliss. 2016. Survey of domain-specific languages for FPGA computing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 1–12. https://doi.org/10.1109/FPL.2016.7577380 ISSN: 1946-1488.

[23] Mika Karaila. 2009. Evolution of a Domain Specific Language and its engineering environment - Lehman's laws revisited. (2009), 7.

[24] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. 2016. Domain-Specific Languages: A Systematic Mapping Study. *Information and Software Technology* 71 (March 2016), 77–91. https://doi.org/10.1016/j.infsof.2015.11.001

[25] J.R. Lewis and B. Martin. 2003. Cryptol: high assurance, retargetable crypto development and validation. In *IEEE Military Communications Conference, 2003. MILCOM 2003.*, Vol. 2. 820–825 Vol.2. https://doi.org/10.1109/MILCOM.2003.1290218

[26] Grischa Liebel, Omar Badreddin, and Rogardt Heldal. 2017. Model Driven Software Engineering in Education: A Multi-Case Study on Perception of Tools and UML. In *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE T)*. 124–133. https://doi.org/10.1109/CSEET.2017.29 ISSN: 2377-570X.

[27] Sandra Macià, Sergi Mateo, Pedro J. Martínez-Ferrer, Vicenç Beltran, Daniel Mira, and Eduard Ayguadé. 2018. Saiph: Towards a DSL for High-Performance Computational Fluid Dynamics. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 (RWDSL2018)*. Association for Computing Machinery, Vienna, Austria, 1–10. https://doi.org/10.1145/3183895.3183896

[28] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. *Comput. Surveys* 37, 4 (Dec. 2005), 316–344. https://doi.org/10.1145/1118890.1118892

[29] Leandro Nascimento, Daniel Viana, Paulo Neto, Dhiego Martins, Vinicius Garcia, and Silvio Meira. 2012. A Systematic Mapping Study on Domain-Specific Languages.

[30] Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. 2014. A Survey on Domain-Specific Languages in Robotics. In *Simulation, Modeling, and Programming for Autonomous Robots (Lecture Notes in Computer Science)*, Davide Brugali, Jan F. Broenink, Torsten Kroeger, and Bruce A. MacDonald (Eds.). Springer International Publishing, Cham, 195–206. https://doi.org/10.1007/978-3-319-11900-7_17

[31] Nuno Oliveira, Maria João Pereira, Pedro Rangel Henriques, and Daniela Cruz. 2009. Domain specific languages: a theoretical survey. *INForum'09 - Simpósio de Informática* (2009). https://bibliotecadigital.ipb.pt/handle/10198/1192 Accepted: 2009-10-01T12:52:37Z Publisher: Faculdade de Ciências da Universidade de Lisboa.

[32] Ildevana Poltronieri, Allan Christopher Pedroso, Avelino Francisco Zorzo, Maicon Bernardino, and Marcia de Borba Campos. 2021. Is Usability Evaluation of DSL Still a Trending Topic? In *Human-Computer Interaction. Theory, Methods and Tools*, Masaaki Kurosu (Ed.). Vol. 12762. Springer International Publishing, Cham, 299–317. https://doi.org/10.1007/978-3-030-78462-1_23 Series Title: Lecture Notes in Computer Science.

[33] Ildevana Poltronieri, Avelino Francisco Zorzo, Maicon Bernardino, and Marcia de Borba Campos. 2018. Usa-DSL: usability evaluation framework for domain-specific languages. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18)*. Association for Computing Machinery, Pau, France, 2013–2021. https://doi.org/10.1145/3167132.3167348

[34] Simon Schauss, Ralf Lämmel, Johannes Härtel, Marcel Heinz, Kevin Klein, Lukas Härtel, and Thorsten Berger. 2017. A chrestomathy of DSL implementations. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017)*. Association for Computing Machinery, Vancouver, BC, Canada, 103–114. https://doi.org/10.1145/3136014.3136038

[35] Mathijs Schuts, Marco Alonso, and Jozef Hooman. 2021. Industrial experiences with the evolution of a DSL. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Domain-Specific Modeling*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/3486603.3486774

[36] Krysta M. Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q#: Enabling scalable quantum computing and development with a high-level domain-specific language. *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018* (2018), 1–10. https://doi.org/10.1145/3183895.3183901 arXiv: 1803.00652.

[37] Marcel van Amstel, Mark van den Brand, and Luc Engelen. 2010. An exercise in iterative domain-specific language design?. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE) on - IWPSE-EVOL '10*. ACM Press, Antwerp, Belgium, 48. https://doi.org/10.1145/1862372.1862386

[38] Tijs van der Storm and Sebastian Erdweg. 2015. Proceedings of the 3rd Workshop on Domain-Specific Language Design and Implementation (DSLDI 2015). *arXiv:1508.03536 [cs]* (Aug. 2015). http://arxiv.org/abs/1508.03536 arXiv: 1508.03536.

[39] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices* 35, 6 (June 2000), 26–36.

https://doi.org/10.1145/352029.352035

[40] Markus Voelter, Bernd Kolb, Klaus Birken, Federico Tomassetti, Patrick Alff, Laurent Wiart, Andreas Wortmann, and Arne Nordmann. 2019. Using language workbenches and domain-specific languages for safety-critical software development. *Software & Systems Modeling* 18, 4 (Aug. 2019), 2507–2530. https://doi.org/10.1007/s10270-018-0679-0

[41] David Wile. 2004. Lessons learned from real DSL experiments. *Science of Computer Programming* 51, 3 (June 2004), 265–290. https://doi.org/10.1016/j.scico.2003.12.006

Appendix E

Static Secrecy Guarantees for Dynamic Condition Response Graphs

Static Secrecy Guarantees for Dynamic Condition Response Graphs (Working Paper)

Holger Stadel Borum Mads Frederik Madsen Søren Debois

A Dynamic Condition Response (DCR) graph models a business process at a high abstraction level. In a distributed setup, A DCR graph lets adversarial actors cooperate in a well-defined, explicit, agreed-upon business process. In this form of cooperation, an actor must leak some information on what activities they perform. However, it is non-obvious to figure out exactly what information is leaked through cooperation. Still, an actor may wish that internal choices in conducting activities remain secret to other actors. Therefore, an actor should be able to reason about what information they leak when agreeing to cooperate in a business process. While an exponential model transformation can be used to analyse secrecy, we seek to leverage the high abstraction level of DCR graphs to obtain a more computational feasible approximation of secrecy. In this paper, we present our ongoing work on this secrecy approximation that is inspired by information flow analysis.

1 Introduction

Actors with conflicting interests can use business process modelling to facilitate cooperation in a well-defined agreed-upon process [1]. For example, a client and a contractor can define their business process using a Dynamic Condition Response (DCR) graph to ensure there is no ambiguity about who is allowed to (or must) perform an activity. Before participating in a business process, an actor may wish to ensure that they do not leak information on their internal choice of activities that could be used against them by an adversarial collaborator. Therefore, we seek an approximation of secrecy in DCR graphs that is both sound and efficient while being complete enough to be useful. Such approximation could also be used in non-adversarial multi-actor collaboration where regulations such as GDPR require enforcing privacy policies.

The contributions of this working paper are:

- A formalisation of what it means to approximate secrecy in DCR executions.
- A presentation of our static secrecy approximation of DCR graphs in its current state.
- The definition of minimal runs in DCR graphs and a proof showing that minimal runs are a sound approximation of runs. We believe the concept of minimal runs is a stepping stone toward proving our approximation is sound.

Furthermore, in the final paper, we seek to:

- Prove the secrecy approximation to be sound, which may require some adjustments to the approximation.
- Improve the secrecy approximation to be more complete, i.e., consider *condition* relations and give fewer false negatives when analysing secrecy.

2 Notation

- $\mathcal{P}(\_)$ denotes the powerset function.
- $\in$ denotes a) set membership, b) subformula membership, e.g., $\top \in \top \vee \bot$ and $\top \notin \bot \wedge \bot$, and c) sequence membership, e.g., $A \in C \cdot A$ and $A \notin C \cdot D$.

3 Background

In the background section, we focus on presenting the concept of DCR graphs. We will also later use a concept of knowledge similar to the modal logic described in [2] but present it in a later section. First, we present a formal definition of DCR graphs and their semantics. Second, we describe what it means for a DCR graph to be distributed among different actors. Third, we define what we mean for an actor to observe the behaviour of a DCR graph.

Definition 1 (Dynamic Condition Response (DCR) graphs [3])**.** A DCR graph is a tuple $G = (\mathtt{E}, \mathtt{M}, \mathtt{R}, \mathtt{L}, l)$, where

- $\mathtt{E}$ is a set of *events*.
- $\mathtt{M} = (\mathtt{M_{Ex}}, \mathtt{M_{In}}, \mathtt{M_{Pe}}, \mathtt{M_{En}}) \in \mathcal{P}(\mathtt{E})^4$ is the *marking* of the graph. Each of the four sets that constitutes a marking denotes, respectively, the events that are *executed*, *included*, *pending*, and *enabled*.
- $\mathtt{R} = (\rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\div)$ is the *relations* of the graph consisting, respectively, of the *conditions*, *responses*, *includes* and *excludes*, where all relations are binary event relations, i.e., in $\mathtt{E}^2$.
- $\mathtt{L}$ is the *labels* of the graph.
- l is the *labelling function* of a graph, mapping events to labels: $l : \mathtt{E} \to \mathtt{L}$.

We denote the set of the possible markings in the graph by $\mathscr{M}(G) \subseteq \mathcal{P}(\mathtt{E})^4$. For a given event e, we denote the set of events it has outgoing include relations to, i.e., $\{e' \in \mathtt{E} \mid e{\rightarrow}{+}e'\}$, as $e{\rightarrow}{+}$. Similarly, we use ${\rightarrow}{+}e$ to denote the set of events e has ingoing include relations from. We use the same notation for the other kinds of relations $\rightarrow\bullet$, $\bullet\rightarrow$, and $\rightarrow\div$.

Definition 2 (Semantics of a DCR graph as a labelled transitions systems [4])**.** Let $G = (\mathtt{E}, \mathtt{M}, \mathtt{R})$ be a DCR graph ($\mathtt{L}$ and l omitted). The corresponding labelled transition system (LTS) is the tuple $T(G) = (\mathscr{M}(G), \mathtt{M}, \mathtt{E}, \rightarrow)$, where $\rightarrow \subseteq \mathscr{M}(G) \times \mathtt{E} \times \mathscr{M}(G)$ is the transition relation given by $\mathtt{M} \xrightarrow{e} \mathtt{M}'$ s.t.

- $\mathtt{M} = (\mathtt{M_{Ex}}, \mathtt{M_{In}}, \mathtt{M_{Pe}}, \mathtt{M_{En}})$ is the marking before the transition
- $e \in \mathtt{M_{En}}$
- $\mathtt{M}' = (\mathtt{M'_{Ex}}, \mathtt{M'_{Pe}}, \mathtt{M'_{In}}, \mathtt{M'_{En}})$ is the marking after the transition, where
  - $\mathtt{M'_{Ex}} = \mathtt{M_{Ex}} \cup \{e\}$
  - $\mathtt{M'_{In}} = (\mathtt{M_{In}} \cup e{\rightarrow}{+}) \setminus e{\rightarrow}{\div}$
  - $\mathtt{M'_{Pe}} = (\mathtt{M_{Pe}} \setminus \{e\}) \cup e{\bullet}{\rightarrow}$
  - $\mathtt{M'_{En}} = \{e' \in \mathtt{M'_{In}} \mid ({\rightarrow}{\bullet}e' \cap \mathtt{M'_{In}}) \subseteq \mathtt{M'_{Ex}}\}$

We call a sequence of transitions in an LTS for a run r and denote all runs of $T(G)$ by $R(G)$. When G is clear from the context, we denote its set of runs by R. For a set of runs R, we denote their final markings by $\mathcal{M}(R)$.

Definition 3 (DCR distribution)**.** A DCR graph can be distributed among a non-empty set of agents $\mathtt{Act}$ s.t. each agent can observe and execute a subset of $\mathtt{E}$. We denote agent a's set of events by $\mathtt{E}_a$. For completeness, we require that the all events is owned by an agent, i.e., $\bigcup_{a \in \mathtt{Act}} \mathtt{E}_a = \mathtt{E}$.

Definition 4 (Observation on a run)**.** An actor a can observe a transition t using some observation function $\delta(\mathtt{E}_a, t)$. The actor can observe a run by lifting δ to $\Delta(\mathtt{E}_a, r)$ by applying δ pointwise to each transition in r and removing any element from the sequence if it is a chosen empty observation value, e.g., $\emptyset$ or ϵ.

In Definition 9, we define the concrete observation function used in this paper. For now, we remain with an unspecified general one. The observation function is both used to describe what an actor may observe and what an actor may not observe. We say that two runs are indistinguishable to an actor iff there is no observational differences between them.

Definition 5 (Indistinguishability of runs)**.** Two runs $r, r' \in R$ are indistinguishable to agent a iff

$$\Delta(\mathtt{E}_a, r) = \Delta(\mathtt{E}_a, r')$$

By $[r]_a$, we denote the equivalence class of runs indistinguishable from r to a in R. By $R^{\backsim a}$, we denote the set of all equivalent classes induced on R by Δ, i.e., $R^{\backsim a} = \{[r]_a \mid r \in R\}$. The indistinguishability relation of runs is trivially an equivalence relation by equality of sequences of sets.

4 Knowledge in Distributed DCR Graphs

We will work with two knowledge perspectives appearing when actors cooperate in DCR graphs. First, *state knowledge* describes the properties of the static image of the system in terms of markings. Second, *action knowledge* describes the properties of the changes to the system in terms of transitions. There is a duality between these knowledge perspectives in that if one knows the changes that have occurred in a system, then one also knows something about the

current state of the system. We need these two perspectives precisely because of this duality in that we will use an actor's *action knowledge* to describe their *state knowledge*.

Definition 6 (Knowledge atoms)**.** Let e be an event in $\mathtt{E}$ then knowledge atoms and their meaning for the action and state perspective are defined as follows:

	action atoms	**state atoms**	
$e\div$	has been made	is	excluded
$e+$	has been made	is	included
$e!$	has been made	is	pending
$e\checkmark$	has been made	is	executed
$e\circ$	has been made	is	enabled
$e\bullet$	has been made	is	disabled
$e\neg!$	has been made	is	not pending
$e\neg\checkmark$	~~has been made~~	is	not executed

We use σ and $e*$ as meta-variables to range over action and state atoms.

Using these knowledge atoms, we now define state knowledge and action knowledge. State knowledge is defined by the function $\rho(\_)$, which returns a set of state atoms that characterises the marking as a static image. The set contains four state atoms per event that describes whether the event is contained in $(\mathtt{M_{Ex}}, \mathtt{M_{In}}, \mathtt{M_{Pe}}, \mathtt{M_{En}})$.

Definition 7 (State knowledge in a marking)**.** Given a marking $\mathtt{M}$ of some DCR graph, let the state knowledge be:

$$\rho(\mathtt{M}) \triangleq \bigcup_{e\in\mathtt{E}} \left\{ \begin{Bmatrix} e+ & e \in \mathtt{M_{In}} \\ e\div & \mathtt{o.w.} \end{Bmatrix}, \begin{Bmatrix} e\checkmark & e \in \mathtt{M_{Ex}} \\ e\neg\checkmark & \mathtt{o.w.} \end{Bmatrix}, \begin{Bmatrix} e! & e \in \mathtt{M_{Pe}} \\ e\neg! & \mathtt{o.w.} \end{Bmatrix}, \begin{Bmatrix} e\circ & e \in \mathtt{M_{En}} \\ e\bullet & \mathtt{o.w.} \end{Bmatrix} \right\}$$

We lift $\rho(\_)$ to $\mathrm{P}(\_)$ that works on a set of markings M by applying ρ to each marking. That is: $\mathrm{P}(M) = \{\rho(\mathtt{M}) \mid \mathtt{M} \in M\}$.

From an action perspective, knowledge is defined by the function $\delta(\_)$ that characterises an LTS transition with an action atom for each marking change occurring in the transition.

Definition 8 (Action knowledge in a transition)**.** Given a transition $\mathtt{M} \xrightarrow{e'} \mathtt{M}'$ let its action knowledge be:

$$\begin{aligned} \delta(\mathtt{M} \xrightarrow{e'} \mathtt{M}') \triangleq\ & \{e+ \mid e \in (\mathtt{M'_{In}} \setminus \mathtt{M_{In}})\} \cup \{e\div \mid e \in (\mathtt{M_{In}} \setminus \mathtt{M'_{In}})\} \\ & \cup \{e! \mid e \in (\mathtt{M'_{Pe}} \setminus \mathtt{M_{Pe}})\} \cup \{e\neg! \mid e \in (\mathtt{M_{Pe}} \setminus \mathtt{M'_{Pe}})\} \\ & \cup \{e\circ \mid e \in (\mathtt{M'_{En}} \setminus \mathtt{M_{En}})\} \cup \{e\bullet \mid e \in (\mathtt{M_{En}} \setminus \mathtt{M'_{En}})\} \\ & \cup \{e\checkmark \mid e \in (\mathtt{M'_{Ex}} \setminus \mathtt{M_{Ex}})\} \end{aligned}$$

We are now ready to define the specific observation function we use in this paper instead of the general used in Definition 4. We say that an actor may observe all the changes that happen to their events in a transition as a set of action atoms. Therefore, the empty set $\emptyset$ is the special empty observation of the observation function.

Definition 9 (Observation function)**.** Given a transition t and an observation set $\mathtt{E}_a$, let the observation of a transition be:

$$\delta(\mathtt{E}_a, t) \triangleq \bigcup_{e \in \mathtt{E}_a} \{e* \mid e* \in \delta(t)\}$$

Definition 10 (Collapsed run observation)**.** For an observation set $\mathtt{E}_a$ and a run r, we use ∇ to collapse a sequence of knowledge in a run observation by

$$\nabla(\mathtt{E}_a, r) = \bigcup_{t \in r} \delta(\mathtt{E}_a, t)$$

By R_∇, we denote a set of runs collapsed with ∇ applied to each run individual run that is $R_\nabla = \{\nabla(\mathtt{E}, r) \mid r \in R\}$. By $R_\nabla^{\backsim a}$, we denote the equivalence class of R collapsed with ∇ that is $R_\nabla^{\backsim a} = \{\{\nabla(\mathtt{E}, r') \mid r' \in R'\} \mid R' \in R^{\backsim a}\}$. We use w as a variable for the members of R_∇ and $R_\nabla^{\backsim a}$ with the mnemonic of a member representing a possible *world*.

We define composite knowledge formulas inductively using conjunction ($\wedge$) and disjunctions ($\vee$). We do not include *negation* for two reasons: First, from a state perspective, we already have built-in negation atoms. E.g., the negation of an event being *included* is that it is *excluded*. Second, from an action perspective, it is difficult to interpret what it means for an event to not occur. If the negation of becoming *included* is becoming *excluded*, then we already have action atoms to model this meaning. If negation means that at some point in time, an event does not occur, then the negation of an atom is almost universally true. Still, we consider whether we can meaningfully include negations to be able to reason about, for example, implications. However, we leave these considerations for future work.

Definition 11 (Composite knowledge in possible worlds)**.** By the judgement $K \models \varphi$, we denote that a predicate φ is known in the knowledge base K whether K is state or action knowledge. K is a set of sets of action or state knowledge:

$$\begin{aligned} K \models \sigma &\text{ iff } \forall\, w \in K \,.\, \sigma \in w \\ K \models \varphi_1 \wedge \varphi_2 &\text{ iff } K \models \varphi_1 \text{ and } K \models \varphi_2 \\ K \models \varphi_1 \vee \varphi_2 &\text{ iff } K \setminus K' \models \varphi_1 \text{ and } K' \models \varphi_2 \end{aligned}$$

Definition 12 (Secrecy of knowledge)**.** A formula φ is secret in a knowledge base K iff $\neg(K \models \varphi)$.

Definition 13 (State secrecy to an agent)**.** A formula φ is secret state knowledge to agent a iff φ is not revealed in any class of indistinguishable runs.

$$\forall R' \in R_{\backsim a} .\, \neg\Big(\mathrm{P}(\mathcal{M}(R')) \models \varphi\Big)$$

Definition 14 (Action secrecy to an agent)**.** A formula φ is secret action knowledge to agent a iff φ is not revealed in any class of indistinguishable runs.

$$\forall R' \in R_{\backsim a} .\, \neg\Big(R'_\nabla \models \varphi\Big)$$

In the introduction of this section, we described a duality between *state knowledge* and *action knowledge*. We formalise a property of this duality by showing that *action secrecy* under certain conditions implies *state secrecy*.

Theorem 4.1 (Secrecy of action atoms implies secrecy in markings)**.** If all state atoms of formula φ is not true in the initial marking $\mathtt{M}_0$ and it is action secret to an agent, then it is also marking secret to the agent. Formally:

$$\forall \sigma \in \varphi \,.\, \sigma \notin \rho(\mathtt{M}_0) \;\wedge\; \forall R' \in R_{\backsim a} \,.\, \neg\Big(R'_\nabla \models \varphi\Big) \implies \forall R' \in R_{\backsim a}. \neg\Big(\mathrm{P}(\mathcal{M}(R')) \models \varphi\Big)$$

We choose to show the more general implication:

$$\forall R' \subseteq R \,.\, \forall \sigma \in \varphi \,.\, \sigma \notin \rho(\mathtt{M}_0) \;\wedge\; \neg\Big(R'_\nabla \models \varphi\Big) \implies \neg\Big(\mathrm{P}(\mathcal{M}(R')) \models \varphi\Big)$$

Proof. Choose any R' and proof the contraposition by assuming $\mathrm{P}(\mathcal{M}(R')) \models \varphi$.
We show $\exists \sigma \in \varphi \,.\, \sigma \in \rho(\mathtt{M}_0) \vee \Big(R'_\nabla \models \varphi\Big)$ by structural induction on $\mathrm{P}(\mathcal{M}(R')) \models \varphi$.

Case 1 $\mathrm{P}(\mathcal{M}(R')) \models \sigma$**.**
By law of excluding middle assume $\forall \sigma \in \sigma \,.\, \sigma \notin \rho(\mathtt{M}_0)$, i.e., σ is not part of the initial marking. For contradiction assume $\neg(R'_\nabla \models \sigma)$, but that means that for some run r in R' then σ did not occur. However, this contradicts that σ is true in all markings of R'.

Case 2 $\mathrm{P}(\mathcal{M}(R')) \models \varphi_1 \wedge \varphi_2$**.**
By IH we know $\exists \sigma \in \varphi_1 \,.\, \sigma \in \rho(\mathtt{M}_0) \;\vee\; R'_\nabla \models \varphi_1$.
By IH we know $\exists \sigma \in \varphi_2 \,.\, \sigma \in \rho(\mathtt{M}_0) \;\vee\; R'_\nabla \models \varphi_2$.
By law of excluding middle assume $\forall \sigma \in \varphi_1 \wedge \varphi_2 \,.\, \sigma \notin \rho(\mathtt{M}_0)$.
The negation of this assumption trivially lets us proof our goal.
$\forall \sigma \in \varphi_1 \wedge \varphi_2 \,.\, \sigma \notin \rho(\mathtt{M}_0)$ contradicts both $\exists \sigma \in \varphi_1 \,.\, \sigma \in \rho(\mathtt{M}_0)$ and $\exists \sigma \in \varphi_2 \,.\, \sigma \in \rho(\mathtt{M}_0)$.
So it must be that $R'_\nabla \models \varphi_1$ and $R'_\nabla \models \varphi_2$.
Using these we construct $R'_\nabla \models \varphi_1 \wedge \varphi_2$.

Case 3 $\mathrm{P}(\mathcal{M}(R')) \models \varphi_1 \vee \varphi_2$**.**
By IH we know $\exists \sigma \in \varphi_1 \,.\, \sigma \in \rho(\mathtt{M}_0) \;\vee\; (R' \setminus R'')_\nabla \models \varphi_1$.
By IH we know $\exists \sigma \in \varphi_2 \,.\, \sigma \in \rho(\mathtt{M}_0) \;\vee\; R''_\nabla \models \varphi_2$.
By law of excluding middle assume the negation $\forall \sigma \in \varphi_1 \vee \varphi_2 \,.\, \sigma \notin \rho(\mathtt{M}_0)$.
It must be that $(R' \setminus R'')_\nabla \models \varphi_1$ and $(R'')_\nabla \models \varphi_2$ because o.w. we reach a contradiction.
Using these we construct $R'_\nabla \models \varphi_1 \vee \varphi_2$. □

5 Abstract Knowledge Approximation

We introduce the notion of an approximation of an agent's action knowledge. Our goal is an over-approximation of action knowledge. That is: the approximation may state that an actor

knows something they do not know. However, on the contrary, the approximation may not state that an actor do not know something that they does.

The purpose of the approximation is to avoid an exponential time transformation of a DCR graph to an LTS as described in Definition 2. We start by defining what it means for a set of sets of action atoms to approximate a set of runs.

Definition 15 (Run approximation)**.** For a given set of runs R, we denote that A approximates R by:

$$R_\nabla \sqsubseteq A \triangleq \forall w \in A \,.\, \exists w' \in R_\nabla \,.\, w' \subseteq w$$

Theorem 5.1 (Soundness of approximation)**.** Given a set of runs R and an approximation A s.t. $R_\nabla \sqsubseteq A$, then A is a sound approximation of secrecy of any formula φ (or, conversely, it is a complete approximation of knowledge).

$$R_\nabla \models \varphi \implies A \models \varphi$$

Proof. Assume $R_\nabla \models \varphi$, continue by induction on φ.

Case $\varphi = \sigma$.
From $R_\nabla \models \sigma$, we know $\forall w \in R_\nabla \,.\, \sigma \in w$.
From $R \sqsubseteq A$, we know that all sets in A is an extension of a set in R_∇.
Therefore, it must be that $\forall w \in A \,.\, \sigma \in w$ and we can construct $A \models \sigma$.

Case $\varphi = \varphi_1 \wedge \varphi_2$.
From $R_\nabla \models \varphi_1 \wedge \varphi_2$, we know $R_\nabla \models \varphi_1$ and $R_\nabla \models \varphi_2$.
By IH on each of these, we know $A \models \varphi_1$ and $A \models \varphi_2$.
We can now construct $A \models \varphi_1 \wedge \varphi_2$.

Case $\varphi = \varphi_1 \vee \varphi_2$
From $R_\nabla \models \varphi_1 \vee \varphi_2$, there must exist an R_2 s.t. $R_1 = R \setminus R_2$, $R_{1\nabla} \models \varphi_1$, and $R_{2\nabla} \models \varphi_2$.
We construct A_1 and A_2, s.t. $A_1 = A \setminus A_2$, $R_1 \sqsubseteq A_1$, and $R_2 \sqsubseteq A_2$.
Let $A_1 = \{a \in A \mid \exists w \in R_{1\nabla} \,.\, w \subseteq a\}$. Per definition we have that $R_1 \sqsubseteq A_1$.
Let $A_2 = A \setminus A_1$.

We show that $R_2 \sqsubseteq A_2$.
For contradiction assume the negation $R_2 \not\sqsubseteq A_2$ which means $\exists w_2 \in A_2 \,.\neg \exists w \in R_2 \,.\, w \subseteq w_2$.
Choose this $w_2 \in A_2$ s.t. there does not exist $w \in R_{2\nabla} \,.\, w \subseteq w_2$.
From $R \sqsubseteq A$, then $\exists w \in R_\nabla \,.\, w \subseteq w_2$ and since w is not in R_2 then $w \in R_1$.
However, this means that $w_2 \in A_1$ which contradicts $w_2 \in A \setminus A_1$.

By IH using the two subtrees with respectively A_1 and A_2, then $A_1 \models \varphi_1$ and $A_2 \models \varphi_2$. We can now construct $A \models \varphi_1 \vee \varphi_2$.

□

6 Concrete Knowledge Approximation

We are going to define our approximation of knowledge as the fixed point of the knowledge expansion function $\mathcal{A}(\_)$ applied repeatedly to an initial approximation of actor knowledge. The idea is to obtain a set of approximations such that for each set of indistinguishable runs, there exists an approximation that approximates it. For terminology, we say that we are building a set of approximations where each approximation consists of different worlds that are indistinguishable to the actor. As the starting point for our fixed point for actor a, we construct a singleton set of approximations that consists of singleton approximation with a world where actor a knows everything about a's events. For simplicity, we do not consider condition relations since we do not believe that they are important for our choice of proof technique, but they make the approximation much more complex.

Definition 16 (Initial knowledge)**.** We define A_0^a to model the initial knowledge for agent a as follows:

$$w^a = \bigcup_{e \in E_a} \left\{ e+, e\div, e\checkmark, e\neg\checkmark, e!, e\neg! \right\}$$

$$A_0^a = \left\{ \{ w^a \} \right\}$$

To introduce ambiguity in the approximation, we define effect equivalent events. The idea is that two events are equivalent iff executing the events has the same effect on the DCR graph. We use this equivalence class in the approximation to say that an actor may not be able to distinguish between the execution of two events only if they belong to the same equivalence class. While this may seem like a severe restriction on the approximation, previous proof efforts have shown that we need something like it to ensure that an actor may not make inferences from the order observations.

Definition 17 (Effect equivalent events)**.** For the purpose of the approximation, we define an equivalence class $[e]$ on events stating two events are equivalent given they have the same outgoing relations.

$$\begin{aligned}[e] = \{ e' \in \mathtt{E} \mid\ & e{\rightarrow}+ = e'{\rightarrow}+ \\ & \wedge\ e{\rightarrow}\div = e'{\rightarrow}\div \\ & \wedge\ e\bullet{\rightarrow} = e'\bullet{\rightarrow} \\ & \wedge\ e{\rightarrow}\bullet = e'{\rightarrow}\bullet \\ & \wedge\ e{\rightarrow}\diamond = e'{\rightarrow}\diamond \}\end{aligned}$$

For a set of events E we let $[E] = \{[e] \mid e \in E\}$.

We are going to define our set of approximations as a fixed point on the approximation expansion function $\mathcal{A}(\_)$ that we will define later. The function consists of three different types of knowledge expansion rules that we define and explain individually. First, $\mathcal{A}_{domain}(\_)$ uses information about an event being disabled to infer that some event could not have been executed. Second, $\mathcal{A}_{\checkmark}(\_)$ uses information about an event being executed to infer the effects of the execution. Third, different versions of $\mathcal{A}_{*_1,*_2,*_3}(\_)$ use state change information to infer what could have caused the change.

We start with $\mathcal{A}_{domain}(\_)$ informally stating that if an approximation contains a world where an event is both disabled and executed, then we add another approximation without this world to our set of approximations.

Definition 18 (Domain restriction)**.**

$$\mathcal{A}_{e\checkmark,e\div}(A) = \{w \in A \mid \exists e \in \mathtt{E}.\ e\checkmark \in w \wedge e\div \in w\}$$
$$\mathcal{A}_{domain}(A_i) = \bigcup_{A \in A_i} \{A \setminus \{w\} \mid w \in \mathcal{A}_{e\checkmark,e\div}(A) \wedge |A| > 1\}$$

The function $W(A, e*)$ lets us pick out all world/event pairs in an approximation where a world contains action atom e$*$.

$$W(A, *) = \{(w, e) \mid w \in A, e* \in w\}$$

The set $\mathcal{A}_{e\checkmark}$ denotes all action atoms one can possibly infer from knowing $e\checkmark$.

$$\begin{aligned}\mathcal{A}_{e\checkmark} = {} & \{e\circ, e+, e\neg!\} \\ & \cup \{e'+, e'\circ \mid e' \in e{\rightarrow}+\} \\ & \cup \{e'\div, e'\bullet \mid e' \in e{\rightarrow}\div\} \\ & \cup \{e'! \mid e' \in e\bullet{\rightarrow}\}\end{aligned}$$

The approximation expansion $\mathcal{A}_{\checkmark}(\_)$ informally says: if there is an approximation with a world that contains an executed event, then create a copy of the approximation with the world replaced with one where all possible inferences are made on that execution.

Definition 19 (Execution forward inference)**.**

$$\mathcal{A}_{\checkmark}(A_i) = \left\{(A \setminus \{w\}) \cup \left\{w \cup \mathcal{A}_{e\checkmark}\right\} \mid A \in A_i, (w, e) \in W(A, \checkmark)\right\}$$

Finally, we need approximation expansions from knowing an event has become included, excluded, or pending. These inferences all follow the same form, so we first present the general form with $*_1$, $*_2$, and $*_3$ as wildcards. Informally, the wildcards say that from knowing $*_1$, you may infer $*_2$ from one of the incoming $*_3$ relations.

Definition 20 (Execution backward inference)**.**

$$\mathcal{A}_{*_1,*_2,*_3}(A_i) = \left\{(A \setminus \{w\}) \cup \left\{w \cup *_2 \mid e' \in [*_3\ e]\right\} \mid A \in A_i, (w, e) \in W(A, *_1)\right\}$$

Using the general form of backwards inference, we can define the following four rules:

- From knowing inclusion, one may infer the execution of one incoming include relation: $\mathcal{A}_{+,\{e'\checkmark\},\twoheadrightarrow+}(\_)$
- From knowing exclusion, one may infer the execution of one incoming exclusion relation: $\mathcal{A}_{\div,\{e'\checkmark\},\twoheadrightarrow\div}(\_)$

- From knowing pending, one may infer the execution of one incoming response relation: $\mathcal{A}_{!,\{e'\checkmark\},\bullet\!\!\rightarrow}(\_)$

Definition 21 (Knowledge expansion function)**.**

$$\begin{aligned}\mathcal{A}(A_i) = A_i \cup \mathcal{A}_{domain}(A_i) \ \cup \ \mathcal{A}_{\checkmark}(A_i) \\ \cup \mathcal{A}_{+,\{e'\checkmark\},\rightarrow\!\!+}(A_i) \ \cup \ \mathcal{A}_{\div,\{e'\checkmark\},\rightarrow\!\!\div}(A_i) \ \cup \ \mathcal{A}_{!,\{e'\checkmark\},\bullet\!\!\rightarrow}(A_i)\end{aligned}$$

Definition 22 (Knowledge fixed point)**.** We let A_{fix} denote the fix point of repeatedly applying the knowledge expansion function to the initial knowledge.

$$A^i_{fix} = \mathcal{A}(...(\mathcal{A}(A^i_0)))$$

Hypothesis 6.1 (Soundness of approximating secrecy)**.** We hypothesise that the secrecy approximation is sound (or that the approximation of knowledge is complete), i.e. for any φ:

$$\forall A \in A^i_{fix} \,.\, \neg(A \vDash \varphi) \implies \forall R \in R_{\backsim i} \,.\, \neg(R_{\nabla} \vDash \varphi)$$

By Theorem 5.1 it suffices to show that:

$$\forall R \in R_{\backsim i} \,.\, \exists A \in A^i_{fix} \,.\, R \sqsubseteq A$$

We do not have proof of this property at the time of writing.

7 Minimal Runs as a Stepping Stone to Proof of Soundness

In working towards a proof of the soundness of the approximation (Hypothesis 6.1), we have found that it is difficult to reason directly on sets of indistinguishable runs. The problem is that in a set of indistinguishable runs, many things may have happened that an actor cannot know anything about. Therefore, we have found it useful to introduce a notion of *minimal runs* that reduces a set of runs to runs that only contain a transition if it affects a later transition or is observable.

Definition 23 (Minimal Run)**.** The run r is minimal to actor a iff all transitions of r are either directly observable to a or enable a later transition in the run. That is, for any division of $r = r' \cdot t \cdot r''$ either:

1. $\delta(\mathtt{E}_a, t) \neq \emptyset$
2. or, $\exists t' \in r''$ s.t. either
   (a) $e+ \in \delta(\mathtt{E}, t) \wedge e\checkmark \in \delta(\mathtt{E}, t')$

Part (2) of the definition of minimal runs is ready to be expanded to the introduction of condition relations with other cases, e.g., (*b*) $e\checkmark \in \delta(\mathtt{E}, t) \wedge e'\checkmark \in \delta(\mathtt{E}, t') \wedge e{\rightarrow}{+}e'$.
We use the notion of a minimal run to select only minimal runs from a larger set of runs.

Definition 24 (Minimal runs)**.** For a set of runs R we denote its subset of runs that are minimal to a by $\lfloor R \rfloor_a$. That is:

$$\lfloor R \rfloor_a = \{r \in R \mid r \text{ is minimal to } a\}$$

It is easy to see that $\lfloor R \rfloor_a \subseteq R$.

The idea of introducing the notion of minimal runs is to use it to prove that our approximation of secrecy is sound. The definition allows us to use the properties stated in Definition 23 instead of trying to prove something about a set of runs where the only known property is that they are indistinguishable.

Lemma 7.1 (Minimal runs characterizes approximations)**.** Given an approximation A and a set of runs R, if A approximates $\lfloor R \rfloor_a$, then it also approximates R.

$$\lfloor R \rfloor_{a\nabla} \sqsubseteq A \implies R_\nabla \sqsubseteq A$$

Proof. Trivial since $\lfloor R \rfloor_a \subseteq R$. That is, if there for all elements in A exists an element in $\lfloor R \rfloor_a$ with some property, then this element also exist in R. □

8 Related Work

To our knowledge, this is the only static analysis of secrecy for DCR graphs. However, secrecy and privacy have been studied in other similar settings of business process modelling. Different work seeks to add privacy and secrecy to different business process models but does not consider the potential information leak of the process itself. For example, Pullonen et al. [5] extend the Business Process Modeling Notation (BPMN) with privacy constructs to model, e.g., confidentiality and privacy. Saleem et al. [6] enhance BPMN with confidentiality, non-repudiation, and integrity constructs. Similarly, Sang and Zhou [7] add privacy and secrecy constructs and secrecy indicators to BPMN.

There are more general approaches to reasoning about information leakage of a system. Alur et al. [8] defines the concept of privacy-preserving refinements in labelled transition systems, which is similar is somewhat similar to our approximation relation. However, our approximation is not a transformation and is knowledge preserving. Still, one could consider making actual knowledge-preserving transformations on a DCR graph as an alternative approach to ours. Information flow [10] analysis has been used to reason about secrecy in general-purpose programs, and while our approximation approach is directly inspired by information flow analysis, the technique is not straightforward applicable. Finally, from a probabilistic secrecy perspective, quantitative information flow [9] is used to bound information leakage but requires a more probabilistic knowledge setup.

9 Conclusion and Future Work

In this working paper, we have presented our work towards providing static secrecy guarantees for Dynamic Condition Response (DCR) graphs. First, we have introduced *state knowledge*

and *action knowledge* as two different knowledge perspectives and shown a relation between them. Second, we have defined the abstract notion of an approximation of knowledge (and thereby secrecy) and presented our current concrete approximation. Finally, we have introduced the concept of minimal DCR runs that may serve as a further knowledge approximation and a stepping stone for a proof approximation soundness.

We are pursuing creating proof of soundness, performing performance experiments, and improving the quality of the approximation as three avenues of future work. We first seek a proof since this process may uncover problems in our approximation that invalidates any subsequent efforts. While we have versions of the approximation that is both more complete and efficient, we found them difficult to reason about. Therefore, we prefer starting by working with a simpler approximation. We have put effort into creating a proof of soundness and have so far been successful in showing a previous version of the approximation to be sound for an actor making a single observation. Although we thought that this proof would allow us to "simply" add an induction step and be done, we found a problem when making this step. What we have found to be difficult is that, on the one hand, a run state that a sequence of events has happened forwards in time, and, on the other hand, our approximation reasons about inferences backwards in time.

References

[1] M. F. Madsen, M. Gaub, T. Høgnason, M. E. Kirkbro, T. Slaats, and S. Debois, "Collaboration among adversaries: distributed workflow execution on a blockchain," in *Symposium on Foundations and Applications of Blockchain*, 2018, p. 8.

[2] R. Fagin, J. Y. Halpern, Y. Moses, and M. Vardi, *Reasoning About Knowledge*, Jan. 2004. [Online]. Available: https://direct.mit.edu/books/book/1825/Reasoning-About-Knowledge

[3] S. Debois, T. Hildebrandt, and T. Slaats, "Concurrency and asynchrony in declarative workflows," in *International Conference on Business Process Management*. Springer, 2016, pp. 72–89.

[4] T. T. Hildebrandt and R. R. Mukkamala, "Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs," *Electronic Proceedings in Theoretical Computer Science*, vol. 69, pp. 59–73, Oct. 2011, arXiv:1110.4161 [cs]. [Online]. Available: http://arxiv.org/abs/1110.4161

[5] P. Pullonen, J. Tom, R. Matulevičius, and A. Toots, "Privacy-enhanced BPMN: enabling data privacy analysis in business processes models," *Software and Systems Modeling*, vol. 18, no. 6, pp. 3235–3264, Dec. 2019. [Online]. Available: https://doi.org/10.1007/s10270-019-00718-z

[6] M. Saleem, J. Jaafar, and M. Hassan, "A domain-specific language for modelling security objectives in a business process models of soa applications," *AISS*, vol. 4, no. 1, pp. 353–362, 2012.

[7] K. S. Sang and B. Zhou, "BPMN security extensions for healthcare process," in *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*. IEEE, 2015, pp. 2340–2345.

[8] R. Alur, P. Černý, and S. Zdancewic, "Preserving Secrecy Under Refinement," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, Eds. Berlin, Heidelberg: Springer, 2006, pp. 107–118.

[9] M. S. Alvim, K. Chatzikokolakis, A. McIver, C. Morgan, C. Palamidessi, and G. Smith, *The Science of Quantitative Information Flow*, ser. Information Security and Cryptography. Cham, Switzerland: Springer, Springer Nature, 2020.

[10] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977. [Online]. Available: https://doi.org/10.1145/359636.359712