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.

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

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

[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.