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, quality assurance

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* [28]. Although co-design is mentioned partially as a rebranding of *participatory design* [6] [28], 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 [31] [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 [28], and similarly, software product lines [4] for DSLs [34] 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 [35] or evaluates modelling tools empirically [27] [32]. 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 [30]. 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 [33] 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, ...)                          Function application
    | if(e, e, e)                        Conditional
    | e.x                                Projection
    | e : {t, ...}                       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, ...)                       Procedure call
    | s ...                              Block

o ::= CashFlow | Reserve

d ::= action x(x : τ, ...)with{s}        Action
    | fun x(x : τ, ...) = e              Function
    | data t < extends t >               Data
      {x : τ <, output as o >, ...}
    | 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 general 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.

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

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

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] [29], 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.

	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`		...

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`.

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. They must check whether values differ from what they are expected to be and experiment with how changes in values affect the calculation. Table 2 shows a debugging spreadsheet of an execution of the example program. 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 [26] 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

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

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

$$\mathsf{E}[\![\mathsf{e}_1.\mathsf{x}]\!](\Gamma) = \mathsf{MAL(e_1.x)}$$

$$\mathsf{E}[\![\mathsf{f(e_1,\ldots,e_n)}]\!](\Gamma) = \begin{cases} [] +_v \mathsf{f'(e'_1,\ldots,e'_n)} +_h c_1 +_h \ldots +_h c_n & \text{if } ss(\mathsf{f}) = \mathsf{f'} \\ MAL(\mathsf{f(e_1,\ldots,e_n)}) +_h c_1 +_h \ldots +_h c_n & \text{if } \mathsf{f} \notin dom(ss) \end{cases}$$

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

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

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

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

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

$$\mathsf{S}[\![\mathsf{update\ x\ in\ e\ with\ s_1\ end}]\!](\Gamma) =$$
$$\text{let } [\mathsf{v}_1, \ldots, \mathsf{v}_n] = \mathsf{E}_{MAL}[\![\mathsf{e}]\!]$$
$$\text{let } \Gamma_1, cs_1 = \mathsf{S}[\![\mathsf{s}_1]\!](\Gamma)$$
$$\text{let } cs'_1 = \mathsf{v}_1\ \mathsf{as\ string} +_v cs_1$$
$$\vdots$$
$$\text{let } \Gamma_n, cs_n = \mathsf{S}[\![\mathsf{s}_1]\!](\Gamma)$$
$$\text{let } cs'_n = \mathsf{v}_1\ \mathsf{as\ string} +_v cs_n$$
$$\text{let } \Gamma, cs'_1 +_h \ldots +_h cs'_n$$

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* [29]. 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.

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

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.

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

We use the environment $\Gamma \in \mathsf{env}$ to keep track of where local variables are placed in cells. Here `env` is of type `string` → `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.

There are several threats to validity that practitioners who seek to transfer our experiences to similar design situations should be aware of.

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. We did not experience such negotiations, possibly because we appeared as neutral academics.

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

This work was supported by the Innovation Fund Denmark within the project ProBaBLI (7076-00029B). We thank 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.

[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.

[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. ISSN: 2153-0866.

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer, Berlin, Heidelberg.

[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.

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

[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. 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.

[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. ISSN: 1092-8138.

[13] J. Steve Davis. 1996. Tools for spreadsheet auditing. *International Journal of Human-Computer Studies* 45, 4 (Oct. 1996), 429–442.

[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. ISSN: 1050-4729.

[15] Bent Flyvbjerg. 2006. Five Misunderstandings About Case-Study Research. *Qualitative Inquiry* 12, 2 (April 2006), 219–245. 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. 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.

[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.

[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.

[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. 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.

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

[26] Don Norman. 2013. *The Design of Everyday Things: Revised and Expanded Edition* (revised edition ed.). Basic Books, New York, New York.

[27] Parsa Pourali and Joanne M. Atlee. 2018. An Empirical Investigation to Understand the Difficulties and Challenges of Software Modellers When Using Modelling Tools. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, Copenhagen Denmark, 224–234.

[28] Elizabeth B.-N. Sanders and Pieter Jan Stappers. 2008. Co-creation and the new landscapes of design. *CoDesign* 4, 1 (March 2008), 5–18.

[29] Peter Sestoft. 2014. *Spreadsheet Implementation Technology: Basics and Extensions*. MIT Press, Cambridge, MA, USA.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.