A Domain-Specific Language for Payroll Calculations: an Experience Report from DATEV

Markus Voelter, Sergej Koščejev, Marcel Riedel, Anna Deitsch and Andreas Hinkelmann

Abstract We review our experience developing a domain-specific language at DATEV, a large payroll service provider. The language enables business programmers to efficiently implement, test, and validate payroll calculations independent of downstream deployment considerations. It is fundamentally functional and addresses core domain challenges such as versioning of calculation rules and data and the processing temporal data. We evaluate the language regarding reduction of complexity in payroll programs, the impact on quality, its suitability for use by domain experts as well as the integration into the IT infrastructure. The chapter concludes with general learnings about building business DSLs.

Key words: Business DSLs, Domain-Specific Languages, End User Programming, Case Study, Jetbrains MPS

Markus Voelter
independent/itemis, e-mail: voelter@acm.org

Sergej Koščejev
independent/itemis, e-mail: sergej@koscejev.cz

Marcel Riedel
DATEV e.G., e-mail: marcel.riedel@datev.de

Anna Deitsch
DATEV e.G., e-mail: anna.deitsch@datev.de

Andreas Hinkelmann
DATEV e.G., e-mail: andreas.hinkelmann@datev.de

1 Introduction

Over the last three years, DATEV, a leading German payroll services provider, has been developing a domain-specific language (DSL) for expressing the calculation logic at the core of their payroll systems. The goal is to allow business programmers to express and test the calculations and their evolution over time in a way that is completely independent of the technical infrastructure that is used to execute them in the data center. Business programmers are people who are experts in the intricacies of the payroll domain and its governing laws and regulations (LaR), but not in software development. This leads to interesting tradeoffs in the design of the DSL. The specific set of challenges that motivated the development of the DSL are given in Sec. 3.2. Payroll might seem dull and not too complicated ("just a bunch of decisions and some math"). However, the need to work on data that changes over time, to follow the evolution of the LaR, and to keep the language understandable for non-expert programmers makes it interesting from a language design perspective. The need for execution independent of the deployment infrastructure in the data center and on other devices plus the required flexibility in terms of service granularity and packaging into user-facing applications add interesting non-functional challenges.

In this chapter we evaluate the development of a real-world, non-trivial DSL targeted at business programmers. We describe the language and tool support in Sec. 5. The specific research questions we address are given in Sec. 4.1 and we answer them in Sec. 6. We also list general learnings from this language development project in Sec. 7 and briefly discuss validity (Sec. 8) of our experience report as a case study. We end the chapter with related work in Sec. 9 and a conclusion (Sec. 10).

2 Terminology

Business Programmers Business programmers understand the (payroll) domain very well but have no formal computer science training. They do not necessarily have a thorough understanding of advanced computer science concepts, such as polymorphism, inheritance, separation of concerns or interface vs. implementation. Nor do they aspire to: they consider themselves experts in the business domain, not in software engineering. They are more than analysts though, because they use (domain-specific) languages to develop a formal, executable representation of the business logic in the domain, and not just (more or less) informal requirements documents.

Domain-Specific Languages We assume that the reader is familiar with domain-specific languages and their productivity benefits (if not, check out [21, 6, 18, 5, 26, 49, 23, 25, 22, 46]). Many DSLs are built for developers; several examples are given by van Deursen et al. in [40]. DSLs targeted at domain experts and business programmers are less common and less well documented. This is probably because they are targeted at narrower domains and often considered non-publishable intellectual property, so the companies that use them successfully consider them as a competitive advantage. The language discussed in this chapter is one of these. We discuss others in Sec. 9.

Modeling vs. Programming The style of DSLs described in this paper can best be seen as a mix between modeling and programming. This is why we use the terms *model* and *program* interchangably. From modeling we borrow high-level domain-specific concepts, while we also use low-level, more generic expressions (for example, for arithmetics, comparisons or decisions) that are traditionally associated with programming. We also mix tabular and diagrammatic notations (modeling) with textual syntax (programming). Finally, we rely on non-trivial type systems, another feature that is not typically associated with modeling. See [43] for a more detailed discussion of this perspective.

3 Context

3.1 Business Context

DATEV is one of Germany's largest software companies and IT service providers with over 8,000 employees and a yearly revenue of more than 1.1 billion EUR. As Germany's leading provider of payroll services for small and medium-sized businesses, DATEV's software and services process more than 13 million payroll slips each month.

DATEV has been developing payroll applications for decades; traditionally the applications ran on a Windows PC, accessing data stored centrally in a data center; we describe some details in Sec. 3.3. Recently, DATEV has decided to make the application available as a web service, shifting the business logic into the data center, and using the browser as the UI. Because the existing implementation cannot be retargeted easily, this decision prompted a rewrite of the whole system.

3.2 Business Challenges

In addition to bringing the payroll system to the cloud, the overall goal is to increase development efficiency (the effort to implement new features, the effort to maintain the software and the time to market) in order to keep up with business needs and the market. More specifically, DATEV faces the following business-critical challenges:

C1 **Externally-Driven Evolution** Every year, changes in the applicable laws and regulations prompt changes to calculations and data structures. Keeping track of these versions reliably is key for business success: versions for previous years remain valid and must remain executable to reproduce old payroll calculations; avoiding duplication of unchanged code sections and the ability to reliably identify and then remove now unnecessary code is important to keep maintenance efforts low.

C2 **Business Agility** The current payroll application is general-purpose: it comes with forms that allow users to enter *all* the data required by the LaR. Several simpler, more use-case specific applications exist as well, with *separate* implementations of (parts of) the same business logic. As DATEV expects an increasing variety of such applications in the future, it becomes crucial that the core calculation logic can be reused (and modularly extended) in new contexts, giving more flexibility to DATEV as to how to make their core assets available to the market. Several of these application require that calculations are precise to the day instead of using the month as the smallest distinguishable unit of time; the current infrastructure cannot provide this granularity.

C3 **Deployment Flexibility** DATEV expects technology changes to be more frequent in the future. By default, calculations will run as a set of microservices in the data center, implemented in Java. In addition, parts will also have to be run in the browser for quick validation and to improve UX, which requires an implementation in JavaScript. And some of the new use case-specific applications mentioned in C2 might be implemented in Swift, for execution on iOS.

3.3 The Legacy System

The original, to-be-replaced system, which has been in use for 20+ years, consists of 25,000 data field definitions and 2.5 million lines of COBOL code. Some of that code is generated from a declarative specification that describes how to load and prepare the data necessary for each calculation. In particular, it describes how to handle the temporal changes of the data, because in the calculation itself, the data is pre-aggregated per month. So if, for example, an employee's salary changes from 5,000 to 6,000 EUR in the middle of a month, the calculation code sees only one number, in this case the latter one. This is a fundamental limitation of the legacy system that must be removed with the new system; we referred to this as "calculations must be precise to the day" above. The implementation of the business logic comprises around 200,000 of the 2.5 million LoC and is expressed using a restricted, easier-to-understand subset of COBOL that relies mostly on `MOVE`, `COMPUTE`, `IF...ELSE` and `DO...UNTIL`, plus the aforementioned specifications for data loading. The specification is around 10 times less code than the “expanded” COBOL.

Throughout this chapter, and in particular in Sec. 5 where we describe the new DSL, we compare to the old system in paragraphs labelled with **LEGACY** .

Kind	**Language**	**# of concepts**
DATEV-proprietary	`payroll.dsl.core`	166
	`payroll.dsl.test`	32
	`payroll.dsl.institution`	26
	`payroll.dsl.migration`	7
Developed for DATEV and then open sourced	`kernelf.datetime`	27
	`kernelf.temporal`	24
Use as-is from KernelF	KernelF	120
	Total	**402**

Table 1 Size of the languages developed for DATEV.

3.4 Why a DSL

In 2017 DATEV started developing a Java protoype for the new system based on Java EE technology. However, this prototype focused on the calculation logic and the deployment into microservices; it did not convincingly address the core challenges of the domain given in Sec. 3.2. As a consequence, and influenced by DATEV’s history with code generators and DSLs (such as the COBOL specifications mentioned above), DATEV management comissioned a prototype DSL as a potentially more efficient path. 22 days were spent on the DSL prototype by itemis. It successfully demonstrated that the key complexity drivers, temporal data, precision-to-the-day and versioning (see Sec. 5) can be significantly simplified using a DSL; this result prompted the development of the full DSL.

3.5 Language Development

Tools The DSL is implemented with the JetBrains MPS language workbench[1] and makes use of utilities from the MPS Extensions repository[2] and

[1] `https://www.jetbrains.com/mps/`

[2] `https://github.com/JetBrains/MPS-extensions`

the mbeddr.platform[3]. The language is based on KernelF [42], a functional programming language implemented in MPS for the express purpose of being embedded in (and extended for) DSLs. All ingredients – except the final payroll DSL – are open source software.

Scope In order to manage the overall risk and organizational impact, the DSL's scope was limited to the core domain logic; in particular, the automation of the integration with the execution infrastructure (beyond the generation of Java code that implements the business logic) was out of scope. However, this decision will likely be revisited in the future, because some related artifacts (such as service interfaces or database schemas) are closely related to the structure of the calculations described with the DSL and could presumably be generated easily.

Size of the Language The overall solution for DATEV is split up into several MPS-level languages, some of them reused from KernelF. Table 1 shows the sizes of these languages. The total of ca. 282 custom language concepts[4] (402 minus those reused from KernelF) makes it a typical DSL in our experience. For example, the PLUTO language described in [46] is in the same ballpark. Our style of language design [43] leads to a relatively large number of first-class concepts.

Effort During the initial development period, four developers, one product owner and a scrum master were involved from DATEV. itemis supplied three language engineers. In total, we spent around 3 person years for analysis, discussions, language design, development and testing of the languages, interpreter, generators and IDE, knowledge transfer on MPS from itemis to DATEV as well as project management. The overall effort of the project (analysis, processes, deployment, management) was more than one order of magniture larger than this – so building a DSL was not perceived as an outlandish investment (but did require persuasion, because the investment would amortize only over time).

Process We used Scrum with two-week sprints. Code was maintained in Git, issues were tracked in GitLab. The DATEV and itemis teams worked at separate locations but met regularly every few weeks.

3.6 System Architecture

The DATEV IT organization has a clear architecture strategy for new applications. They rely on microservices, each with a well-defined business context, developed by applying the Domain-Driven Design [14] practices to focus on the domain logic while keeping deployment as simple as possible. To ease

[3] `https://github.com/mbeddr/mbeddr.core`

[4] A concept in MPS is similar to a metaclass in MOF/EMF or a non-terminal in grammar-based systems.

operations in DATEV's data center, applications are required to choose from selected technology stacks. The new payroll application is built on Java within the Spring Cloud ecosystem hosted on a Cloud Foundry Platform-as-a-Service (PaaS) environment. We discuss the details of the architecture and the consequences for the DSL in Sec. 6.4. As we have said before, we do not generate infrastructure integration code from the DSL to limit the scope; we generate Java classes which are then manually integrated into services.

3.7 Current Status of the System

The first production use of the DSL was in late 2018, as part of a simple online payroll calculation tool: this is one of the aforementioned new apps that packages parts of the overall calculation logic into a use case-specific application. Twenty business areas (such as wage tax, social insurance and church tax) have been implemented for two years (2018 and 2019), making use of key language features such as temporal data and versioning.

As of early 2020, all microservices that make up the evolving software-as-a-service payroll application have their core calculation logic generated from DSL models.

Five business programmers currently use the DSL productively, supported by the language development team. Over time, as the legacy application is rewritten step-by-step, dozens of business programmers are expected to work with the language routinely.

The language continues to evolve to reflect the changing requirements, typically, a new language version is released to the end-users every 14 days, based on 14-day Scrum sprints.

Today, development is done mostly inhouse at DATEV with occasional support by itemis, especially if and when extensions or fixes of KernelF are required, or when special skills regarding internals of MPS are needed.

4 Case Study Setup

4.1 Research Questions

Experts in any domain typically point out how complex their domain is; the payroll domain is no exception. The authors hypothesize that much of this perceived complexity is accidental because of a language that is unsuitable to the domain or is induced by too much dependency on the implementation technology: if the right abstractions were used, most of this accidental complexity would vanish, revealing the core of the domain to be much simpler. Our metric for the total complexity (essential + accidental) in this context is the size of programs as well as the amount of work/time required to make typical evolutionary changes to programs; if we can reduce this metric while still solving the domain problem, we have successfully reduced accidental complexity. This leads us to the first research question:

RQ1 Is a suitably-designed DSL able to significantly reduce the perceived complexity in the payroll domain?

The alignment of the DSL with the domain, improved analyzability and tool support, plus the reduced (accidental) complexity of the business logic, should make the models easier to test and validate than code, reducing the potential for errors. The automation through code generation should reduce low-level coding errors. On the other hand, using a DSL introduces additional components into the developer tool chain that could introduce bugs: language workbenches, language definitions, interpreters, generators.

RQ2 Does the use of DSLs and the associated tools increase or decrease the quality of the final product?

Key to increasing flexibility and speed as described in C1 and C2 is to empower the business programmers to express and validate executable versions of the LaR that govern payroll calculation. Once the tests written by the business programmer are green (and assuming they have enough coverage, which we measure), no downstream engineer will ever have to look at the domain logic expressed by business programmers to ensure the business logic is correct[5].

However, for the quality assurance based on tests written by the business programmers to work, the business programmers have to understand the (semantics of the) DSL. This is perceived as a challenge because, in the experience of many engineers, DATEV's business programmers have to learn using abstractions: they prefer small-step, imperative programs. Instead of

```
var entitled = listOfBusinessTrips.distance.sum > 3000
```

they prefer to write

```
var entitled = false
var tripSum  = 0
foreach t in listOfBusinessTrips
   tripSum += t.distance
end
if tripSum > 3000
  entitled = true
end
```

because it is "easier to understand". Clearly, if the very minor step of abstraction involved in the first alternative is "too much", then a DSL with even higher-level abstraction is a challenging proposition.

[5] Of course they look at the code as part of their maintainance of generators, libraries and frameworks, but not for routine domain development work.

RQ3 Can a DSL that reduces complexity be taught to the business programmers with an effort that is acceptable to them?

The code generated from models has to exist in the context of a data center architecture and other deployment platforms in the future (C3 and Sec. 3.6). Building, testing and packaging the models and the generated code has to fit with the development processes and technology stacks used for the data center, and must also be future-proof.

RQ4 How well does the DSL and its use for application development fit with established IT development processes and system architecture?

In this chapter we do not analyze the effort of building the language because we have shown in previous papers [49, 46] that the effort for language development based on MPS is in the same ballpark as the development of many other software artifacts and can be carried by development projects; the efforts for the current project were comparable.

4.2 Data Collected

For this case study we collected numbers and statistics from the code base at the time of writing. We revisited the commit history and the issues in our bug tracker to remind us about the evolution of the languages. The personal recollections of the development team were an additional input.

5 The Language and Tooling

In this section we describe the core features of the DSL[6]. These features have been designed to reduce the complexity of the programs to the essential core of the domain (RQ1) and to reduce the risk of particular kinds of errors (RQ2), while at the same time being accessible to business programmers (RQ3) and keeping the models independent of a particular execution paradigm or target platform (RQ4); in the rest of this section we do not highlight this connection for each specific feature.

[6] The language uses German keywords; in the screenshots, we have translated them to English to ease understanding.

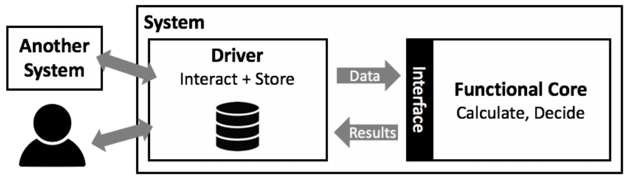


Fig. 1 Overall architecture of the calculations and the driver.

5.1 A Functional Language

The language is functional: values are immutable and variables are assigned exactly once. The individual calculations are small. This fits well with a domain that is heavy in calculations and decisions, programs are easy to analyze, and it allows us to reuse many low-level non-DSL component that orchestrates calculations (decide what to calculate in which order) and handles state (supply data and context). Fig. 1 illustrates this structure.

LEGACY The COBOL system was imperative (making local analysis harder), allowing read-and-write access to a catalog of global data from everywhere. Understanding which data was modified where was hard, and the execution order was fixed.■

An additional drawback of the functional approach is that it was not easy to teach it to the business programmers, especially treating values as immutable and assign-once semantics for local variables. The challenge was particularly pronounced for those who have extensive experience in the old COBOL system, a fully imperative world.

5.2 Execution

A continuous integration (CI) pipelines generates the DSL programs (and all tests) to Java code, packages it, stores it in a Maven repository for deployment, and also runs all tests. End users never invoke this generator; they work with an interpreter that is directly integrated into the IDE (see Fig. 6).

5.3 High-Level Structure

DATEV's payroll domain comprises several dozen business areas such as `tax`, `health insurance`, or `company car`. The DSL uses these as its top-level structure. Business areas then contain `versions` (see Sec. 5.9), versions contain `modules` (think of them as files), which in turn contain the declarations for types, data structures and calculations.

5.4 Domain Data Types

In addition to the usual primitive types, the language supports dates, currency and percentages as built-in types, in terms of syntax, type system, operators and IDE support.

```
val taxRate   : %%%  = 20%            // type explicitly given
val minIncome : EUR  = 2000 EUR
val minTax           = 200 EUR        // type inferred
val deadline         = /2019 01 01/

fun calcTax(income: EUR, d: date)
  = if d > deadline                   // compare dates
      then if income > minIncome      // compare currency
             then (taxRate of income) // work with percentages
             else minTax + 10 EUR     // calculate with currency
      else 0 EUR
```

Users can define record-like data that serves as input and results for calculations (see `data` and `result` in Fig. 3).

LEGACY Data was defined in a GUI tool called Datenkatalog; COBOL structures were generated from these definitions. A changing requirement for the business logic required coordinated changes in the Datenkatalog, the COBOL code and in the user-facing Windows UI, which made implementing straight-forward requirements cumbersome and error-prone.■

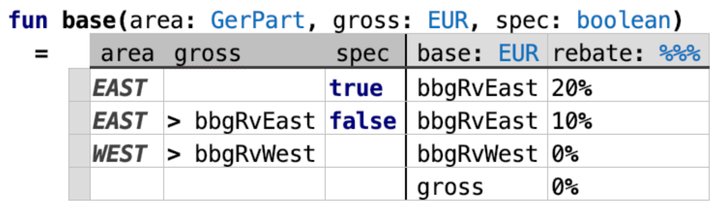
5.5 Tables

The calculations in our domain are characterized by lots of special cases that are the result of years of (not always systematic) evolution of the LaR. Capturing these concisely is infeasible with the `if-then-else` or `switch-case` constructs commonly found in general-purpose languages. Instead we have developed various forms of decision tables (Fig. 2).

LEGACY The requirements documents created from analyzing the law already contains decision tables. However, there was no direct way to represent them in the COBOL sources.■

5.6 Temporal Data

Most data changes over time: employment contracts begin and end, salaries vary, a person's marital status changes. The language has types for temporally evolving data, and the operators (`+`, `-`, `*`, `/`, `&&`, `||`, `>`, `>=`, `<`, `<=`, `==`, `!=`) are overridden to work as expected, but returning temporal values as well (see Fig. 4). There is also support for reducing a temporal type `TT<U>` to a value



```
fun base(area: GerPart, gross: EUR, spec: boolean)
 =
```

area	gross	spec	base: EUR	rebate: %%%
EAST		true	bbgRvEast	20%
EAST	> bbgRvEast	false	bbgRvEast	10%
WEST	> bbgRvWest		bbgRvWest	0%
			gross	0%

Fig. 2 A decision table in a function. Decision tables have one or more columns to match (here: columns 1 through 3) and return one or more values (4th and 5th column). Note the use of "abbreviated expressions" for comparison (2nd column) and the implicit equaliy comparison (columns 1 and 3) They are executed top-down in order to allow a special case to preempt a more general one.

of type `U`, for example by selecting the last value in a month or by computing a monthly weighted average.

Consider the following example. The function takes two temporal values `salary` and `religiousAffil`; "temporal" means that each argument represents a value that varies over time. In the first line, the function reduces this temporal value to a primitive value of type `EUR` by grabbing the last entry in the month `m` in the time series represented by `TT<religion>`. It then checks if that value is `catholic` or `protestant`. If so, it computes the weighted average of the `salary` within `m`.

```
fun churchTax(salary: TT<EUR>, religiousAffil: TT<religion>, m: month) =
  // last religious affiliation reported in any month m is relevant
  if religiousAffil.reduce(LAST in m).isIn(catholic, protestant)
    // use weighted average salary in that month
    then salary.reduce(WEIGHTED in m) * 0.15
    else 0 EUR
```

Some language constructs, such as calculation rules, define an implicit temporal context (typically the `month` for which a result is calculated; see Fig. 3). At such program locations the `@` shortcut invokes the default reduction that is declared for the data (see the `data`s in Fig. 3). Temporal data, and the shortcut syntax in calculation rules are a major enabler for the day-specific calculations mentioned as part of C2.

LEGACY All temporal aspects were handled in the generated data access layer. The smallest granularity of data was "one number per month"; a day-granular treatment of data, as described above, was not supported.■

A second dimension of time At its core, the system is bitemporal [20]: the first temporal dimension D_1 captures the points in time where values change; it is represented by the `TT` types discussed above. A second dimension D_2 represents the point in time from which we look at the data. For example, in May 2019 (D_2), a user enters that on April 1 2019 (D_1) the salary changes to 6,000 EUR. However, in June (D_2), this is corrected to 5,500 EUR. So, depending on whether we look at the data from May or June (D_2), the salary changes on April 1 (D_1) to 6,000 or 5,500 EUR. To keep complexity under control, we have decided that D2 will be handled outside the language, as part of the driver application that runs the calculation and supplies the data. We do not cover the driver or D_2 in the chapter.

5.7 Indexing

Many data items are indexed with a particular employee or client. Similarly, lots of data is time-indexed. Indexing means that they are implicitly associated with one or more keys such as the employee or month. The DSL provides direct support for indexing; for example, in Fig. 3, `TaxReport` is indexed with the built-in abstraction employee (`[EM]`) and time-indexed (`[monthly]`). This means that each employee in the system has one instance of `TaxReport` per month.

In the `calculation` rule (right part of Fig. 3) the created `TaxReport` is automatically associated with the current month and the context employee, both implicitly provided by the driver application. Together with the `@`-notation that invokes the default reduction, this makes for a notably compact syntax.

Understanding the difference between temporal data (a *continuous* value that changes at arbitrary points in time) and time-indexed data (exactly one instance *per* month or year) was a major conceptual breakthrough during the development of the language.

```
data EmployeeData [EM] {
  religion : TT<religion> reduce LAST
}

data EmploymentRel [EM] {
  salary  : TT<EUR> reduce WEIGHTED
  endDate : date
}

result [monthly] TaxReport [EM] {
  salaryInMonth     : EUR
  incomeTax         : EUR
  relTax            : EUR
} where [ incomeTax + relTax < salaryInMonth]
```

```
calculation for TaxReport
    depends emp : EmployeeData
            rel : EmploymentRel
      where rel.endDate >= increment.begin
as [monthly] {
  salaryInThatMonth := rel.salary@
  incomeTax := salaryInMonth * INCOME_RATE
  val mustpay = emp.religion@.isIn(cath, prot)
  relTax := alt[ mustpay   => rel.salary@ * REL_RATE ]
               [ otherwise => 0€                      ]
}
```

Fig. 3 Data definitions, results and calculation rules. `EmployeeData` is a data structure that is indexed with the built-in domain entity `EM` (short for employee). It has one attribute whose type is a temporal type whose default reduction rule is to use the last value a given period. `EmploymentRelationship` is similar, but the salary of an employment, when reduced to a simple value, is weighted over the period. `TaxReport` is a result data structure, which means it is calculated by a rule. That rule is shown next, its signature expressing that it consumes the two `data` enties and produces the `TaxReport`. The rule is tagged `monthly`, which means that during execution, there's an implicit context parameter, referred to by `increment`, that is of type `month`. When the default reduction operator `@` is invoked on a temporal value, it is this month that is used as the period within which the default reduction is applied.

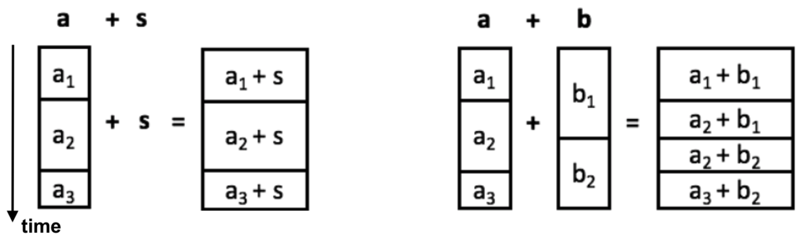


Fig. 4 Adding a scalar `s` to a temporal value `a` means that the `s` is added to each of the slices. Adding two temporal values `a` and `b` "reslices" the result and adds the components. Similar for other arithmetic operators.

5.8 Declarative Dependencies

Calculations specify the input data and upstream results they depend on (see the `depends` clause in Fig. 3). For indexed data the dependency implicitly refers to data with the same indexes (in Fig. 3, the current month and employee). It is also possible to "navigate" time explicitly by using special operators (such as `-1` or `allInCurYear`) in square brackets; these unify simple dependencies with temporal aggregation.

```
calculation for TaxStatistics
  depends thisMth : TaxReport               // TaxReport from this month (implicit)
          lastMth : TaxReport[-1]           // TaxReport from last month
          rest    : TaxReport[allInCurYear] // list of reports
as [monthly] { ... }                        // from all prev. months
```

LEGACY Dependencies were indirect by accessing the same global data. While analysis tools could extract business area dependencies through analysis of global data access, this required a separate tool and was cumbersome. Unintended dependencies resulted.■

5.9 Versioning

The versioning described in the remainder of this section covers the domain-related evolution of the models, mostly driven by changing LaR. However, currently, during development, much of the evolution of the code for a business area is driven by changing technical requirements and the increase of domain coverage, where, for example, a data structure now has additional attributes because the business area covers additional special cases (for example, the wage tax for people who are both employed *and* have a freelance side business).

This often leads to breaking changes in the data structures and the need for downstream business areas to adapt. To manage these changes, all our models are of course versioned in git. To coordinate semantically dependent changes, we use organizational mechanisms (discussing and planning changes with all involved business areas) supported by the IDE, for example, by highlighting those data structures that are in fact shared by different business areas.

Domain-related Versioning Challenge C1 states that programs must evolve with the LaR, but previous models and their tests must remain valid to reproduce old payroll calculations. Using the version control system to represent this evolution does not work because several versions would have to be checked out simultaneously. Instead, the DSL supports the notion of a `version` of a business area. A version specifies the date at which it becomes valid.

Version Kinds Ideally, the code written for the various business areas can evolve independently from others. While internal changes to calculations do not affect other business areas, interface changes requires coordination. To make this explicit, the language supports two kinds of versions that are distinguished by how they differ from its predecessor: a `calculation` version may change parameter values and the logic inside `calculation`s. And an `interface` version differs in the structure of the data that is exchanged with other business areas.

The kinds also govern how new versions affect the versions of downstream business areas and the requirement for redeploying generated code. A `calculation` version requires regeneration and redeployment because the code that performs the calculation of the result changes. However, since the data structures (which act as the interface between business areas) remain stable, downstream business areas are unaffected, structurally, and no redeplyoment is required. In addition, since calculations depend on data and not on the particular rule that computes it (e.g., the dependency on `TaxReport` above), calculations that span different `calculation` versions are automatically dispatched; we explain how this works below.

An `interface` version is more invasive. The new version can completely remove a data or result entity and optionally declare a new one that (semantically) replaces the deleted one. Downstream business areas then have to create a new version as well to make use of this new data structure. Alternatively, entities can also evolve. For one, new members can be added; in addition existing members can be deleted. A deleted member can no longer be accessed in the new version, but it is retained in the physical data structures that serve as the technical interface to external clients in order to retain backward compatibility. Renaming or changing the type is not supported, because the impact would be harder to track, both for the tool and also for humans.

Note that the IDE knows about these versions and their start dates: code completion and type checking will exploit the information statically to provide version-aware IDE support.

Inter-Version Reuse A new version inherits all contents (parameters, data structures, calculation rules) from its predecessor; the new version can then selectively override. Parameters replace similarly-named parameters; a parameter can also be deprecated, which means that it cannot be used in the new (and even newer) versions. For `data` and `result` structures, the new version can chose whether to adds fields to, or remove fields from the old one, as we have explained above.

To understand the automatic dispatch of rule invocations for a version, let's consider the following example. It shows three successive versions of some business area, with several rules for a particular result data structure `Res`.

```
version 1 (from 2018)     version 2 (from 2019)        version 3 (from 2020)
----------------------    -------------------------    --------------------------
add r1(Res, c1)  {...}    change r1(Res, c1) {...}
add r2(Res, c2)  {...}                                 change r2(Res, c2New) {...}
add r3(Res, c3)  {...}    delete r3
                                                       add r4(Res, c4) {...}
add default(Res) {...}    change default(Res) {...}
```

As we have said before, the language supports several rules for the same result data structure (`Res`) as long as they have different applicability conditions (`c1, c2, ...`); this can be interpreted as several functions with the same signature but different preconditions. The default rule is executed if none of the conditions of the other rules applies. Let's understand this logic by figuring out which rules apply for each year in the example above.

From the perspective of a calculation that applies to 2018, it is rather obvious, because only version 1 is visible. So `v1.r1`, `v1.r2`, `v1.r3` and the `v1.default` rule are candidates for execution; the decision which one actually runs is based on the conditions.

From the perspective of 2019, the effective set of rules is `v2.r1` (because this one has been overridden in version 2), `v1.r2` (it is inherited), and `v2.default` (also overridden). `v1.r3` is not available, it has been deleted in version 2.

From the perspective of 2020, the following rules are candidates for execution: `v3.r4` (added in version 3), `v3.r2` (changed in version 3), `v2.default` (inherited from version 2) and `v2.r1` (also inherited).

To aid understanding which rules are relevant for each particular point in time, the IDE shows the effective set of rules for each version. It is the task of the developer to make sure that the conditions are complete and overlap-free for each year; an integration of an SMT solver to verify this analytically has been a goal for a long time, but has not yet been realized.

The exact semantics of overriding rules in successive versions took time to get right. In an earlier phase of development of the DSL the set of rules for a given result in a version were not named. A new version did not explicitly override a discrete version, it only added another rule with a particular condition; if that condition was similar to a condition in the previous version, the rule with that condition would be effectively replaced. But conditions were also allowed to overlap partially, effectively overriding a rule only for a special case. However, this was too complex for our users to understand, especially considering the lack of SMT support. We have thus moved to the slightly less flexible but easier to understand approach described above.

Removal of Unneeded Code The LaR require DATEV to be able to reproduce payroll calculations for five years. As a corollary, DATEV wants to be able to remove code that is no longer needed for the versions of the preceding five years. Because dependencies are explicit and the type system takes versions into account, the respective analysis can be performed statically and "old" code can be removed with confidence.

LEGACY No language-level abstraction for versions was available, which led to a lot of accidental complexity. One consequence was very little systematic reuse between the code for subsequent "versions", driving up maintenance effort. Systematic detection and removal of unneeded code was not supported, which means that (presumably), dead code was still actively maintained.■

5.10 Data Validation

Before the system can make any calculations, it has to ensure that the input data is valid relative to user-defined rules. The DSL supports validation of single fields (generated to browser-side UI code), of record-style data structures, as well as validations that span multiple records. For the validation of fields and records, the lexical location for the respective validation condition is obvious: at the field and record declarations (see the `where`-clause on `TaxReport` in Fig. 3). For validations that span multiple records is less obvious where to write them down. We decided on the following two locations. For records that have relationships (`Employment --[1..*]--> Car`) the referencing record can navigate into the target as part of its own validation conditions. The other case occurs when a calculation rule uses several structuraly unrelated inputs (so we cannot navigate from one to the other), but still needs to constrain the data relative to each other. In this case, the respective condition is expressed as a precondition of the particular rule (similar to the one in the `where`-clause of the calculation in Fig. 3).

LEGACY Data validation was mostly duplicated in the client application and the COBOL backend, synchronized manually. This was tedious and error-prone.■

5.11 Testing Support

To reach the goal of enabling business programmers to fully validate the domain logic, specific test support has been built on top of KernelF's generic support for unit tests; Fig. 5 shows an example.

Repository The repository specifies the "universe" of data upon which a set of tests are executed using a compact table notation. It contains instances of `data` items (optionally indexed) as well as instances of `result`s. Since the execution semantics of the language is functional and incremental, the engine only calculates those results that are not in the repository; a result that is put into the repository as part of the test setup is effectively a mock for a calculation.

Checks Checks query the system for calculated data. They specify a point in time that determines the applicable version as well the expected result (e.g., `TaxReport(500 EUR, 100 EUR)`). Multiple checks can be run against a single repository.

Deltas and Vectors Often, a set of checks use only slightly different repository setups. The language supports two ways of avoiding the duplication resulting from separate repositories: (1) a check can specify a delta repository that overwrites some of the data in the test-wide repository, and (2) the data in the repository as well as the queried data can contain variables (e.g., `TaxReport(i, c)`). Values for these variables form a test vector. A check can then specify a list of such vectors as a table and run all of them.

LEGACY Because of the sequential nature of the programs and the reliance on global data, unit testing was cumbersome and the developers relied on integration-level tests. Data was supplied through CSV files without tool-supported checking against the required input. It relied on COBOL code, so accidental complexity was higher and the turnaround time was slower.■

5.12 External Components

Although the system described in this chapter is a complete re-development, it does not live in isolation, and integration with external components is necessary. We give two examples here.

External Logic The core of the wage tax calculation in Germany is complicated. Luckily, the German Ministry of Finance provides this calculation as a specification[7] plus an associated XML file. DATEV has developed a generator that transforms this XML file to Java and packages it into a JAR file. To access it from DSL programs, we have implemented a language concept that allows to call this generated library like a built in function to perform the calculation. This language concept also performs static type checking, based on the type specifications in the XML. The same library is also used in the generated microservice, albeit without the DSL wrapper.

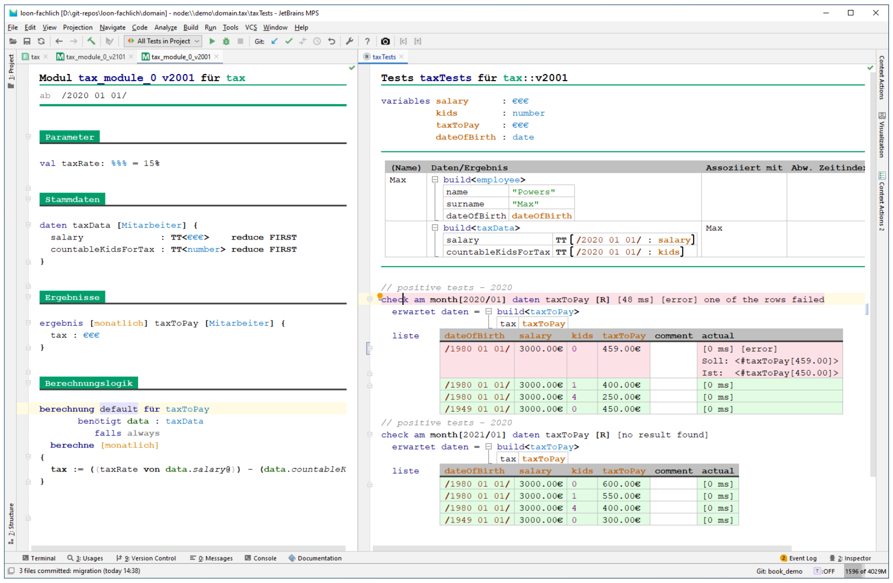
[7] `https://www.bundesfinanzministerium.de/Content/DE/Downloads/Steuern/Steuerarten/Lohnsteuer/Programmablaufplan/2019-11-11-PAP-2020-anlage-1.pdf?__blob=publicationFilev=2`

External Data To calculate the salary slip, we need access to lots of data that is not maintained in our system, but provided to us as JSON files with a known schema; examples include the list of municipal tax offices and the health insurance providers in Germany. We have developed a language extension that allows us to map the JSON structures to data entities declared in models. Based on this mapping, the data is accessible from within DSL programs as if it were native to our system. The mapping exploits advanced features of our language, such as temporality, so even external data that changes over time is accessible to DSL models using the convenient native support for temporal data. And like any other data, we can mock the data in (interpreted) unit tests. Once deployed to the data center, the data is retrieved in realtime via REST calls and made available to the generated implementation of the business logic. In line with the general goal of the DSL, this approach decouples the domain developer from the technical concerns, while at the same time leading to minimal integration effort for the backend developer.

5.13 IDE Features

MPS provides the usual **editor support** for the DSL. Several domain-specific refactorings are available. To visualize the big picture, the **IDE** provides shortcuts for common operations, for example, to navigate to all rules for a particular result, or to the corresponding declaration in previous and future versions. For a test, the IDE shows the transitive closure of the versions of the accessed business areas applicable for the point in time specified in the test. There is a customized project view that shows the hierarchy of business areas, modules, versions, tests and declarations. Several PlantUML[8]-based **visualisations** help illustrate relationships and highlight circular dependencies. The in-IDE **interpreter** allows users to execute tests with zero turnaround time and effort; the checks are color-coded red/green to show success or failure. Tests can also be analyzsed step by step using a **tracer**; every expression execution is shown in a tree, and values of program nodes (usually expressions) can be overlayed over the program code. The infrastructure also supports the measurement of three kinds of **coverage** for a set of test cases: (i) language concepts and the relationships between them; (ii) the interpreter implementation; (iii) test subject program nodes. The data is collected by the interpreter and then visualized in the IDE. Finally, various reports help understand the big picture; for example there are reports for all validation messages used in the system. For a visual impression of the see Fig. 5 and this video: `https://vimeo.com/339303255`

[8] `http://plantuml.com`



```
Modul tax_module_0 v2001 für tax
ab  /2020 01 01/

Parameter

val taxRate: %%% = 15%

Stammdaten

daten taxData [Mitarbeiter] {
  salary             : TT<€€€>    reduce FIRST
  countableKidsForTax : TT<number> reduce FIRST
}

Ergebnisse

ergebnis [monatlich] taxToPay [Mitarbeiter] {
  tax : €€€
}

Berechnungslogik

berechnung default für taxToPay
       benötigt data : taxData
          falls always
  berechne [monatlich]
{
  tax := ((taxRate von data.salary@)) - (data.countableK
}
```

```
Tests taxTests für tax::v2001

variables salary      : €€€
          kids        : number
          taxToPay    : €€€
          dateOfBirth : date
```

(Name)	Daten/Ergebnis	Assoziiert mit	Abw. Zeitinde
Max	build<employee> name "Powers" surname "Max" dateOfBirth dateOfBirth		
	build<taxData> salary TT [/2020 01 01/ : salary] countableKidsForTax TT [/2020 01 01/ : kids]	Max	

```
// positive tests - 2020
check am month[2020/01] daten taxToPay [R] [48 ms] [error] one of the rows failed
  erwartet daten = build<taxToPay>
                     tax taxToPay
  liste
```

dateOfBirth	salary	kids	taxToPay	comment	actual
/1980 01 01/	3000.00€	0	459.00€		[0 ms] [error] Soll: <#taxToPay[459.00]> Ist: <#taxToPay[450.00]>
/1980 01 01/	3000.00€	1	400.00€		[0 ms]
/1980 01 01/	3000.00€	4	250.00€		[0 ms]
/1949 01 01/	3000.00€	0	450.00€		[0 ms]

```
// positive tests - 2020
check am month[2021/01] daten taxToPay [R] [no result found]
  erwartet daten = build<taxToPay>
                     tax taxToPay
  liste
```

dateOfBirth	salary	kids	taxToPay	comment	actual
/1980 01 01/	3000.00€	0	600.00€		[0 ms]
/1980 01 01/	3000.00€	1	550.00€		[0 ms]
/1980 01 01/	3000.00€	4	400.00€		[0 ms]
/1949 01 01/	3000.00€	0	300.00€		[0 ms]

Fig. 5 A screenshot of the MPS IDE with the languages developed for DATEV. On the left you see data and rule definitions, and on the right you can see a corresponding test case.

LEGACY No meaningful IDE support is available. No coverage measurement was available, so the code was manually analyzed to design test cases in a way they would cover the code.■

6 Evaluation

6.1 RQ1 Is a suitably-designed DSL able to significantly reduce the perceived complexity in the payroll domain?

Comparison to the old system Specific differences that led to accidental complexity have been pointed out in the chapter already using the **LEGACY** label. We will not repeat them here.

Three-Layer Separation A first look at the payroll domain suggests that it is mostly about complex decisions and calculations. And indeed, these are a problem worth addressing. For example, Sec. 5 shows how we use decision tables and domain-specific data types to reduce complexity and increase readability. However, most of the domain complexity comes from temporal data, dependencies between business areas and the variability between different versions; hence most language features relate to those. Addressing these complexities directly in the language allowed us to reduce the perceived complexity significantly. At the beginning of the project we heard statements like “this looks simple – why do we need a DSL?” from some of the engineers at DATEV. Of course it is only simple *because* of the DSL. We have seen this three-layer structure – surface logic, hidden complexities, technical aspects – in other domains, too [45].

Debugging The ease of locating and understanding errors is a major factor for productivity and a major pain point in the **LEGACY** system. The DSL brings three improvements: (1) The execution of a calculation collects explanations, end-user relevant messages that explain a potentially non-intuitive result (“The church tax rate is 9% instead of the standard 8% because the person lives in Bad Wimpfen.”). (2) The tracer mentioned above that shows the complete calculation tree with values overlaid over the program. Those two approaches allow the business developer to track down errors without considering technical aspects. Case (3) is different: if a calculation succeeds in the interpreter but fails in the generated Java code, then there is either an error in the interpreter or in the generator; debugging the interpreter implementation or the generated code, together with an engineer, is necessary. But once the infrastructure is tested, this third step is rare and most of the debugging can be done with methods 1 and 2.

Post-Mortem Debugging If the calculation is correct in the interpreter but then fails in the generated Java, the error must lie in the generator, and the problem must be debugged by a technical developer. However, sometimes a corner case might occur in a real-world calculation for which no test exists, leading to a faulty result. To understand this, users can let the tracer create a test case which debugs the calculation in the IDE. Depending on how often this will occur in practice (it shouldn’t, with sufficient test coverage!), we will add functionality to collect the data at runtime and automatically construct a corresponding test case.

6.2 RQ2 Does the use of DSLs and the associated tools increase or decrease the quality of the final product?

Reuse of a Mature Language Reuse is a proven means of reducing development effort and increasing quality. There is lots of research into language modularity and composition [12], and it works robustly in MPS [47]. A low-level functional language is an good candidate for reuse because most DSLs include expressions for arithmetics, comparison and logical operations. KernelF [42] is such as language, and the payroll DSL uses it as its core. KernelF and its interpreter has been used in several other projects and it is therefore stable and mature. In particular, its test suite achives 100% branch coverage regarding the semantics definition in the interpreter. The payroll

DSL benefited significantly; we found only one major semantic bug in KernelF and fixed a few minor issues.

Redundant Execution The duplication of execution semantics in the interpreter and the generator adds complexity, and it took some time to align the semantics of the two by ensuring all tests succeed in both environments. On the other hand, the relatively simpler interpreter acts as a kind of "executable specification" for the more complex generator. Aligning the two was simplified by the fact that both are ultimately Java, so they could share runtime classes (such as `BigInteger`, `BigDecimal` or `Date`), avoiding discrepancies in small-step semantics. We are confident in the approach, because we have used it before in healthcare [46], where the redundancy was essential to the safety argument.

Generated Low-Level Code Because the mapping to the execution infrastructure is generated, it is very easy to achieve consistency in the implementation. A change in the use of the infrastructure, a bug fix, or an optimization requires only a change in the generator to update the whole code base. This approach increases agility regarding the technical aspects of the system. Of course, the generator can also be a source of errors: a mistake in the generator spreads into the code base as well. However, such errors are usually relatively easy to find, because lots of things break simultaneously. Based on our experience in this and other projects, the trade off works: once the generator is tested reasonably well, overall stability increases, and the time to roll out improvements decreases.

Reuse of QA infrastructure We were able to reuse the KernelF infrastructure for testing, including the ability to run interpreted tests on the CI server as well as the facilities for measuring various aspects of coverage for the language implementation.

Multi-Step QA A goal of the DSL is to allow business programmers to express and test the payroll logic without caring about technical aspects (C3). To this end, we separate functional and technical concerns: models contain only business logic, the generators, runtimes and frameworks take care of the technical aspects. Our development process (see Fig. 6) adds concerns step by step, which means that a failure diagnoses precisely where a fault lies. Step (1) concerns and tests the functional correctness. A failing test indicates an error in the business logic or in the interpreter (in early phases of the project while the interpreter is not yet mature). Step (2) translates the business logic to Java and thus concerns performance. We run the same set of tests, and if one fails, either the generator or the interpreter is faulty; likely it is the generator, because it is more complex, and the test has already been considered correct in the previous step. Step (3) adds the infrastructure to make the system scale. A failure after this step indicates a problem with frameworks or the platform.

Documentation and Communication Because the DSL programs are free of technical concerns and use domain-relevant abstractions and notations,

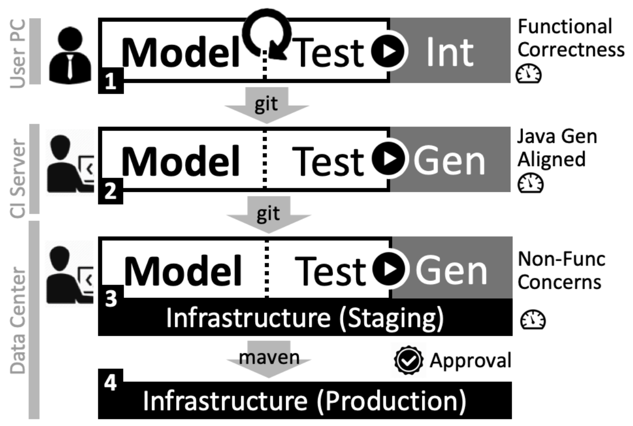


Fig. 6 The process from modeling to deployment. (1) Iteratively model the business logic, test it with the interpreter ("Int" in the figure), measure coverage; (2) Generate ("Gen") to Java on the CI server, run the same tests, measure Java coverage; (3) embed "domain.jar" into the microservices infrastructure and run the same tests in a distributed environment; (4) after final QA and management approval, move the new version from staging to production.

the need for documentation (beyond comments that explain rationales) is greatly reduced. This prevents the documentation from diverging from the code. The language definition and the tests cases also serve as a formalized interface between the business programmers and the technical teams, which puts their communication and coordination efforts on a more solid foundation, reducing the risk of misunderstandings and other inefficiencies.

6.3 RQ3 Can a DSL that reduces complexity be taught to domain-experts in a reasonable amount of time?

IDE Support Users wanted tool support beyond the MPS defaults. For example, they expected buttons to insert `data`, `enum` or `calculation` declarations into a (new version of a) module, intentions to selectively copy declarations inherited from a previous version into the current one for subsequent change, or menu items for creating a test case for a module. While many of these make sense because they bundle repeated multi-step changes, others were *exact* duplicates as the default code completion. For example, typing `calc` and then using code completion produces

```
calculation for <result>
        depends <dependencies>
as [] {
  <code>
}
```

which is what our users wanted a button to do. Once users got familiar with code completion (as opposed to buttons known from classical applications), the requests for these fine-grained UI actions subsided.

Error Checking The quality of analyses and associated error messages is important for the acceptance of the DSL with its users. We put a lot of effort into a precise wording of error messages and into making sure they are reported at the correct locations(s); many error messages come with quick fixes that automatically fix the problem when triggered by the user.

Liveness Short turnaround times help developers stay "in the flow". In addition, for people with limited experience with abstraction such as our users, it is very useful to be able to execute programs immediately and reduce the gap between the program and its execution – which is one of the motivations for live programming [28]. In our architecture, this rapid turnaround is facilitated by the in-IDE interpreter: users iteratively create models, play with them, and then write tests to verify the behavior (see (1) in Fig. 6).

The Big Picture Reuse between versions was a contested issue: a new version `v4` selectively overwrites the declarations from previous versions, requiring the user to look through `v1`..`v3` to understand the effective contents of `v4`. End users did not appreciate this need to mentally assemble "everything" from parts to achieve reuse. To resolve this tension, we exploit MPS' projectional editor to optionally show inherited declarations in the new version: "everything" can be seen in one place, optionally. In addition, we integrated automatically-rendered UML-style diagrams to show the relationships between the declarations in a module, as well as a tree view that shows the applicable versions and their effective declarations for a calculation that spans several business areas. Since each business area can have a different set of versions that might start on different dates, it is not trivial to understand which versions of which business area are applicable for a calculation on some particular date.

End-User Involvement During initial development, involvement of domain experts was difficult. The team worked on the core language abstractions without focussing on usability. User feedback would have been negative for "superficial" reasons; we wanted to avoid such negative first impressions. In addition, many future users struggle with formulating the requirements for the DSL because they are not aware of the design space for the language and IDE. Instead, the DATEV language developers, themselves former "payrollers", acted as proxies for our users. Once the language started to mature, future users were integrated more broadly through demo sessions, screencasts and workshops. The feedback loops were shortend and we focused on more and more detailed aspects of the language and the IDE.

Teaching The best way to teach the DSL is to let future users experience the language. We did this in four steps. (1) Language developers create sample models that address common problems in the domain; (2) These samples form the basis for tutorials, demos, screencasts and howtos that illustrate language and tooling in a way that connects with future users. (3) User/developerpairs implement the examples; and (4) Gradually, users try to independently implement further examples, supporting each other. Language developers are available as 2nd level support. Initially the last step was harder than expected; our users told us that routine work didn't allow them to spend time "playing around with the DSL". Now, after some time of learning, the approach works really well and the business programmers "experiment" with the language as they try to implement new requirements.

Git Most business programmers had not used a version control system before. To keep the complexity of using Git low, we taught the users the basics using the built-in IDE features of MPS, avoiding the command-line. In addition, developers try to avoid merge conflicts (perceived as especially cumbersome) by using a development process that avoids parallel work on the same parts of the system by different developers in the first place.

Infrastructure A crucial ingredient to limiting the complexity for the end users is that they are not required to deal with *any* part of the deployment stack; once they get their tests running in the IDE and have pushed the changes into Git, they are done (see Fig. 6).

LEGACY Developers were required to deal with multiple components of the overall stack, increasing complexity.■

6.4 RQ4 How well does the DSL and its use for application development fit with established IT development processes and system architecture?

Layered Architecture The DSL was specifically scoped to cover only the business logic of the domain; integration with the deployment infrastructure is done on the level of the generated code using agreed interfaces. Before considering a DSL for the business logic, DATEV had already decided to use a microservice architecture and to apply domain-driven design [14]. Each service would be layered like an onion (compare [37]), with outside-in dependencies. Fig. 7 illustrates the current architecture of a microservice focusing on DSL integration. The `domain` layer contains the generated business logic. It relies on libraries that form the `DSL runtime` that are shared among services for the generated DSL code. The `api` layer exposes the service functionality to the outside and, in our case, also contains the `Driver` that provides the current employee, the current date, access to reference data as well as to the (calculation results of) other services. Finally, the `infrastructure` layer contains technology adapters (database, UI, middleware).

Generating the technology-independent `domain` layer from models was a natural integration point for the DSL. The first test of this approach was to remodel, and then regenerate, a manually written `domain` layer for a prototype microservice. Agreeing on the `DSL runtime` interfaces and those

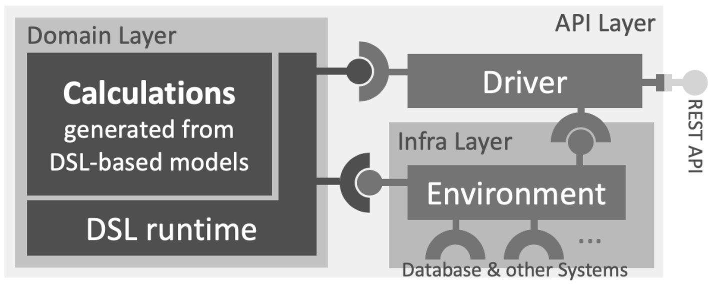


Fig. 7 The adaptation of the onion architecture integrating the DSL approach.

implemented by the generated domain layer took a couple of iterations. In particular, building a common understanding of the relation between versions, their impact on deployment, and an the API that supports cross-version polymorphism for `calculation` versions took time.

Flexible Deployment From corporate architecture guidelines it was clear from the start that the calculations would run in a distributed, microservice architecture. However, the allocation of functionality to services was open because of the different trade-offs regarding performance, scalability, stability and service management overhead: (i) every version of every business area a separate service; (ii) all versions of a business area in one service; (iii) multiple business areas with all their version in one service; (iv) all business areas and versions in one service.

It was useful that the DSL can accommodate all four options by adapting generators or build scripts. In addition, the development of business logic could proceed without deployment decision in the architecture team, which helped to "unblock" the teams. Ultimately option (iii) was chosen for the initial deployment. A different trade-off might lead to choosing different options in the future. For now, the mapping of business areas to services is performed outside of the DSL, as part of the build process.

LEGACY The monolithic COBOL architecture could not be broken up easily into different deployment units, making the trade-offs harder to reevaluate.■

Technology Change Even during the development of the system, the execution infrastructure was changed from JEE to Spring; this required changes to method signatures and annotations in the generated POJOs and the persistence layer. Those changes could be achieved by modifying the generators. No modification of the business logic was necessary. Overall, the integration effort into the new technology stack was low, in line with our expectations and the "promise" of model-driven development, DSLs and code generation.

Execution Paradigm Another technical aspect concerns the execution of the computation. Initially it was not clear whether, when data changes, computations would recalculate everything for a particular employee and month, or whether they would store intermediate results and use the dependencies to incrementally recalculate the transitive closure of the changed data. The functional nature of the language allows both, after generators and runtimes are adapted. Currently, we use the simpler from-scratch approach.

More generally, future optimizations in terms of scalability or resource consumption will very likely be implementable in the generators and frameworks, without invasive changes to the DSL programs.

LEGACY The monolithic COBOL architecture relied on an hard-coded, imperative execution paradigm.■

Technology-independent Testing A natural consequence of the onion architecture is that the domain layer can be run without infrastructure, by mocking the infrastructure interfaces. This is an important ingredient of our QA approach, as illustrated by step 2 in Fig. 6.

Generator Complexity Developing the generator to Java was more effort than expected. One reason was that the functional language had to be mapped to Java's imperative style. This led to excessive use of closures in the generated code as well as long, hard-to-debug chained dot expressions. For the latter we have implemented a transformation that splits the chains into sequences of variable declarations before generation. For example,

```
val v = aContext.longExpr.anEvenLongerOne
        .andAHigherOrderOne(|it.withSomething|)
```

is transformed to

```
val __t1 = aContext.longExpr
val __t2 = __t1.anEvenLongerOne
val v = __t2.andAHigherOrderOne(|it.withSomething|)
```

The generated Java code will then also use a sequence of variable declaration statements, making it easier to read and debug.

DATEV initially wanted the generated code to look exactly as if it were hand-written, partly to simplify debugging, partly to preempt those engineers who were sceptical about code generation, and partly to make the integration with the existing infrastructure, frameworks and programming guidelines easier. We were required to respect naming conventions and use strongly-typed APIs even behind the interfaces to the generated black box. This led to larger, more complex generators (we developed an intermediate language to deal with versioning of strongly-typed APIs) as well as to a significantly bigger (generated) codebase compared to a solution that relied on more generic APIs inside the generated code.

Over time, as more and more of the microservices contain generated business logic and the trust in the generator-based approach increases, DATEV realized that the hard requirements for strongly-typed data structures and readable generated code decreases. As of now, the first microservices process the data structures as JSON and do not rely on strongly typed Java-classes internally. This significantly reduces the complexity of the generator. Another example is that necessay checks, if a new version of a business object still has a value for a deleted field, doesn't lead to a ”compilation error” anymore – we now report this as an error the validation of the model (as opposed to a compile error on code level), which is fully accepted by the users.

Build Process The automated build shown in Fig. 6 had to be integrated into DATEV's CI infrastructure. In principle this is not a problem with MPS – it can be used in headless mode to check, generate and test models. However, the (partially reusable) build infrastructure of KernelF relies on gradle and DATEV required the use of Maven. Also, setting up an MPS headless build is generally tedious and error prone (see [47]). This led to a few weeks of additional effort.

MPS Distribution MPS is a Java application that runs on the desktop. It does require infrastructure for deploying the tool to the (virtualized) PCs of the users. The effort to set this up was higher than expected.

Language Updates Like most other IDEs, MPS relies on a plugin system; the languages and IDE customizations used by business programmers are such plugins. The integration server builds these plugins for every commit, and at the end of each sprint, these are made available to the MPS installations via a CloudFoundry web server. The MPS installations prompt the user to download the new plugins and potentially run model migrations.

7 General Learnings

Specific and Generic A well-designed general-purpose programming language has a small number of orthogonal and composable language concepts that allow users to define their own abstractions. For DSLs, in contrast, it is less important that users can define their own abstractions; instead, users expect the DSL to come with prededefined abstractions for the use cases relevant to the domain (which partially explains the large number of language concepts in Table 1).

However, if a DSL is designed in this rigid way, it cannot grow towards more expressive power over time without expensive structural refactorings. An extensible functional language like KernelF, together with MPS' capabilities, provides an elegant middle ground. Structurally, everything is an `Expression`. However, initial iterations of the language only ship with use-case specific, high-level expressions that are easy for the end users to understand (an example is the Boolean `list l1 and l2 do not share data` used in a constraint). As users become more experienced, one can add more expressive constructs (`l1.intersect(l2).isEmpty`) without changing the fundamental architecture of the language.

A second example: in several cases, our end users asked us to remove genericity in favour of a more specific approach (with better, less generic tool support); for example, when assigning to an `enum`-valued result variable in a calculation, users suggested code completion to propose only the enum literals ("it is too complex otherwise"), not realizing that they might want to *compute* the value. Instead of changing the structure of the language, we used MPS' capability for constraining language concepts at particular locations to only show/allow `enum` literals when computing `enum`-typed values. However, once people realized they did indeed want to compute them, we added them back in without any significant change to the language. Potentially, this filtering can be user-specific, catering to both newbies and more experienced users.

We have since found a great compromise: when assigning to `enum`-valued variables we customized the MPS code completion menu to show the `enum` literals at the top and in bold, and all the other expressions further down; this will highlight the "simple" approach while still allowing more expressive, generic expressions.

There's a saying in the computer science community: "Every DSL will eventually evolve into a general-purpose language". We think this is wrong – this DSL and other similar ones are not a replacement for Java or C. However, most DSLs, as they evolve, will need more (mostly lower level) features that make it Turing complete. But these languages still have lots of domain-specific concepts in them as well, so they are not general-purpose. However, when selecting the tooling to build the languge, make sure you chose one that is expressive enough to be able to handle this evolution.

Functional Programming The functional approach is very useful for lots of technical reasons – such as easy extensibility and relatively simple analyzability – and to provide lots of end-user-relevant features with acceptable implementation effort. However, many business programmers, especially those who have extensive imperative experience, consider it a challenge. We mitigated this by providing high-level declarative abstractions for things that are ubiquituous in the domain, so that "low-level functional algorithmic programming" is required rarely. The approach is (half jokingly) called "funclerative programming" in [42].

The Price of Reuse Language reuse comes at a price: an existing language concept might be 100% what is needed in a DSL. For example, a keyword might be English instead of German, one might prefer a different default (for example, does the type `number` without a specification of decimal digits denote an integer or a real?) or one might prefer the `first` operation on a `list<T>` to be of type `T` instead of `opt<T>` because the reusing language does not use option types. In practice, we usually start a new DSL by reusing (potentially non-ideal) language constructs from KernelF to get the project going quickly and proof its viability. In later phases, once we know the investment will not be wasted, we replace (some of) them with more ideal, custom-developed constructs.

We have also reused the KernelF-to-Java generator; the low-level abstractions, such as the basic expressions, worked in the DATEV context without problems. The higher-level the reused language construct, the more likely it is that the choices made by the original generator developer do not fit with the project-specific context. For example, the generator for messages, a facility for collecting and reporting errors and warnings to the user, did not fit directly. Luckily, MPS provides mechanisms to override the existing generator in such situations.

SMT It turns out that many analyses that are expected by our business programmers require abstract interpretation [9, 10] on an SMT [3] domain. An example is checking a set of Boolean expressions (in a `switch`-like expression or distributed over several `calculation`s) for completeness and overlap. However, users do not necessarily understand why this is so much more complicated than some of the other error checking performed by the IDE. We have observed the same in other DSL projects [46]. To make such analyses possible, we would have to translate all of KernelF to solvers like Z3 [11], and build this transformation in a way that is easily extensible towards constructs from DSLs that extend or embed KernelF. This is a major task; itemis has been working on for the last few years, but has not yet finished.

This can be seen as a negative consequece of reusing KernelF: our language is now so expressive that it is prohibitively expensive to transate it into SMT. However, the domain does require this expressiveness, so *not* reusing KernelF would not make it better. However, what we *can* learn from this is that we should develop a successor to KernelF which is integrated with – meaning: translatable to – an SMT solver right from the start.

Attention to Detail There is different emphasis between end users and language engineers regarding detail. Examples abound. We had to allow leading zeros in date and month literals as well as German umlauts and § signs in identifiers. We developed an infrastructure to manage abbreviations of name components (a central list of allowed name components, component-wise code completion of multi-part names based on the list, checking rules, refactorings to extract name components, showing names in abbreviated and expanded form). We spent a lot of time on the exact rounding rules for currency types. We worked on tool infrastructure to support internationalization for messages and integration with external translation tools. And we were required to use German-language keywords for DATEV-specific language concepts, which leads to a curious mix of German and English, because the keywords of KernelF-concepts cannot easily be changed to German due to an MPS limitation. When initially estimating the overall effort for the project, we did not take such requirements into account.

Pros and Cons of the Projectional Editor A projectional editor is a good fit for DSLs like the current one because of its support for non-textual notations such as tables, the ability to use non-parseable, natural language-like syntax and its support for more highly-structured, text-template-like notations such as the one for `calculation` shown in Sec. 6.2. And since grammar cells [48] have been available, the "feel" of the editor is close enough to a text editor for it to be acceptable to most users. The projectional editor is also an important enabler for the versatile support in MPS for language extension and composition [41], because one never runs into parsing ambiguities.

However, we did run into a few limitations. For example, insertion into the headers of decision tables took a while to get smooth. And the `/yyyy mm dd/` syntax for dates is a non-convincing compromise: `dd.mm.yyyy` would be preferred, but it is ambiguous with decimal number literals because a projectional editor has no look-ahead to be able to distinguish the two.

MPS Limitations More generally, we encountered a few limitations of MPS, including (1) keywords in multiple languages, (2) projecting nodes in places other than their location in the AST, (3) execution of expensive global validations, and (4) execution of a single set of tests both using the interpreter and the Java generator.

For (2), (3) and (4) we developed workarounds, (1) is unresolved. For a thorough discussion of the "good, bad and the ugly" of MPS regarding the development of large-scale DSLs, we refer to [47].

MPS for End Users From an end user perspective, MPS looks and feels too much like an IDE (even though everything that is not needed for the payroll DSL has been removed from the UI). The requirement to install it locally on the users' PCs is also not a plus. An ideal tool would run in the browser and feel more like a modern web app, while still supporting all the language engineering available in MPS. However, as far as the authors know, such a tool is currently not available, even though various communities have started to develop prototypes; examples include Jetbrains' WebMPS[9] and itemis' modelix[10].

MPS Learning Curve Learning to be a productive MPS *language developer* is hard, for many reasons: most developers do not have *language* development experience in the first place, MPS is a powerful tool and has many facets, not everything is as consistent within MPS as it could be, and the documentation is not as thorough and far-reaching as it should be. While the focus of this chapter is not the mechanics of building the DSL (we refer the reader to [47]), it is worth pointing out that it took longer than expected for the new language developers to become productive with MPS.

Resistance to Change? It is often said that domain experts and business programmers resist change, such as when moving to a DSL. We have heard it too in this project. Upon closer inquiry we have found that what they are really saying is something like: "last time we had to change, the new system was not ready yet, bugs were not fixed fast enough, we were not taught how it works, our feedback was not taken seriously, and, during the period of changing to the new system we were expected to be as productive as during

[9] `https://confluence.jetbrains.com/download/attachments/145293472/WebMPS.pdf ?version=1modificationDate=1571311862000api=v2`

[10] `https://modelix.org`

the time before the change". Avoid these things, and you will encounter much less resistance.

8 Validity

Internal Validity The authors were involved in the development of the DSL, which might lead to bias regarding its success. However, we think that we have provided enough evidence in this chapter to allow readers to judge the success of the project for themselves. At the very least, the chapter should be useful to raise concerns that can be considered when starting a new DSL project.

External Validity Can the findings be extrapolated to outside the payroll domain? We think yes. Our experience, and the experience of others [34] with projects in the healthcare [46], finance, tax and public administration domain suggests that domains that have to handle complex decisions and calculations generally benefit from DSLs.

How specific are the results to MPS? The core approach of using DSLs to capture business logic does not depend on MPS; for example, the German federal employment agency uses a set of Xtext-based DSLs to model their enterprise application data, validation, business rules and processes [19]. However, limitations might apply. For example reusing an existing expression language such as KernelF requires support for modular language composition; there is significant variability among language workbenches as to how robustly this is supported [13]. Similarly, MPS' support for non-textual notations is crucial for reducing accidental complexity (consider the decision tables). Other language workbenches, such as Xtext, do not directly support mixed notations. However, in the case of Xtext, they could be built with other Eclipse-based UI technologies, albeit with higher effort and less seamlessly integrated. Summing up, it is less obvious how language workbenches other than MPS could be used to build DSLs like the one discussed in this chapter, though it is definitely possible. For a more general comparison of language workbenches see [13].

Are the results specific to DSLs in general? We compare the DSL approach to a decades-old legacy system. A more meaningful comparison might be with a modern implementation using state-of-the-art programming languages and technologies. Such a comparison is not available. However, there are several indications that a DSL will still come out favourably. First, we showed already during the 22 day proof-of-concept that the DSL can better handle the complexities of the domain such as temporal data and versions. Second, considering the attention to (domain-specific) details by our end users, it is hard to see how a general-purpose language could fullfil them. Third, the enforced separation in the software architecture improved the collaboration of the business programmers and the software developers due to the focus. Finally, the flexibility regarding infrastructure, execution paradigm and performance are hard to achieve without a "custom compiler" that decouples the business logic from its implementation on a particular platform.

Conclusion Validity and Risks An important factor for the success of the new DSL and tooling is the long term maintenance effort. Obviously, at this point in the lifecycle of the DSL, we have only very limited data to draw conclusions about this. However, we can say the following: Regarding the maintenance of the business logic expressed with the DSL, there is reason to be optimistic regarding long-term maintainability, because the language has been designed specifically to address maintenance challenges through its support for versions, temporal data and garbage collection, driven by specific previous experience at DATEV.

Regarding the maintenance of the language implementation itself, the situation is comparable to every other reusable assets, such as a framework or a tool: ongoing effort has to be spent in order to avoid "code rot". A related risks is the dependence on MPS. In the past, DATEV had used proprietary tools and COBOL preprocessors. It turned out to be a challenge to maintain the tools and technololgies over time, leading to a maintenance backlog and thus reduced acceptance of the technology itself. However, DATEV considered this risk acceptable for MPS.

9 Related Work

Business DSLs The closest related work is the use of DSLs for specifying public benefit payments and tax rules by the Dutch Tax Agency [34]: they also target non-programmer users, deal with temporal data and use MPS. Their language uses a syntax that is more natural language-like. The language is in production for specifying the whole Dutch public benefits system and is being introduced in the tax domain. In personal conversations the team reported significant savings in efforts and much shortened time-to-market; unfortunately, no comprehensive case study has been published to date.

The stated goal of Intentional Software [36] was to allow non-programmers to participate directly in the creation of software using DSLs with mixed notations, very much like MPS. Kolk and Voelter [24] give a high-level overview of its prototypical use in an insurance. No detailed case study is available and, although promising, the system never went into production. Intentional Software has since been acquired[11] by Microsoft and as far as we know, the team is no longer working on DSLs and language workbenches.

Risla [39] is a DSL for implementing financial instruments (loan, swap, future). It targeted at engineers, but a questionnaire-style frontend allows financial experts to define new products as well. The paper reports a speed-up from three months to three weeks per product, as well as significant increases

[11] `https://techcrunch.com/2017/04/18/microsoft-acquires-intentional-software-and-brings-old-friend-back-into-fold/`

in readability and portability. Compared to our case, this language is smaller in scope, does not come with IDE support and is used to describe much smaller/simpler programs.

Rebel [38] is used to specify algorithms used in banks (such as money transfers) and then use model checking to verify correctness. The emphasis is on the verification of correctness, and indeed, prototypical used at ING found errors in existing algorithms. However, they also found that the programming-language style syntax was not accessible to non-programmers. This explains why lots of effort goes into syntax of DSLs when targetting non-programmers.

Peyton-Jones introduces a DSL for financial contracts [35, 29]. The language allows the concise expression of contracts: "only 10 combinators are required". However, its reliance on advanced functional programming techniques, its embedding in Haskell and the resulting syntax and limited tool support makes it hard to use for non-programmers. In addition, temporality and versioning were not in scope. A similar language is described in [33], with similar findings. The Actuarial Modeling Language [8] is not implemented in Haskell, but its design makes it clearly targeted at mathematicians, who are used to formal languages and abstraction; the challenge is different to ours.

BPM and Workflows Classical workflow and business process management languages such as Barzdin et al. [4], Faerber et al. [15] or OMG's BPMN [7] capture business processes: which party performs which activity with which data, under which conditions. They do not address the core business logic of, for example, insurance products, tax calculations or payroll. More recently, the OMG has defined the Decision Model and Notation (DMN) [32]. It contains an expression language not unlike KernelF and incorporates various forms of decision tables to make complex decisions more amenable to non-programmers. We could not find published case studies about the use of DMN in industrial practice. Also, the standard (and existing tools) are not extensible to allow adding domain-specific abstractions such as temporal data or versioning. Implementing DMN in MPS in order to to allow such extensibility would be an interesting exercise.

Spreadsheets Many business problems are attacked with Excel. However, when seen through the lens of DSLs, spreadsheets lack a data schema, type checking, the ability to generate code, and a language-aware IDE. While spreadsheets are a good source to mine knowledge that goes into DSLs, we do not consider sophisticated spreadsheets a DSL and therefore do not compare further.

Other Non-Programmer DSLs Several DSL target healthcare: the languages described in Voelter et al. [46] and Florence et al. [17] allow healthcare professionals to describe therapies and dosing medicines, those discussed in Kendrick [1] and Ronald [2] help healthcare researchers simulate epidemiological models and understand the spread and treatment of malaria, respectively. All papers report increased productivity, and better integration of business programmers into the development process; this is in line with our findings.

A wide range of DSL-like languages target engineers in non-software domains; many examples are listed on the Metacase website [30]. Tools like Simulink, ASCET or SCADE are widely used in control-heavy domains. However, their abstractions are so general and low level (state machine or dataflow graphs) that it is a stretch to call them domain-specific; they are better thought of as general-purpose languages for a particular computational paradigm. Their main contribution is a graphical notations (which seems to be liked by engineers) and that they can generate code that has particular safety or performance characteristics (which is relevant in automotive or aerospace).

Teaching Languages Several languages have been designed specifically to teach programming, especially to children; the most well-known are LOGO [16] and Scratch [27]. Obviously, they cannot be used directly in our context because they lack the domain-specific abstractions needed for the payroll domain.

Should these languages be seen as DSLs? LOGO could be seen as a DSL for line drawing. But Scratch is better seen as a general-purpose language that uses a particular notation (the nested shapes). More generally, the design decisions that make these languages accessible to novice programmers are not applicable the same that make them good DSLs: they emphasize generic simplicity and learnability over domain-specificity and productivity. However, these languages could be used to teach the domain experts and business programmers the basics of programming needed to use the DSL effectively.

Other similar initiatives exist, including some teaching programming from scratch [31], some that assume basic knowledge of Excel [44] and in the context of computational thinking [51]. There are also “domain-specific teaching languages”, which help novices to learn programming *in a particular domain*. For example, CoBlox [50] does this for industrial robots. However, CoBlox and similar languages do not attempt to cover the whole domain and scale to real-world problems, which is in contrast to the payroll DSL described in this chapter.

DSLs for Programmers DSLs are widely used as a tool for software developers; as we have mentioned in the introduction, most report significant productivity boosts. However, since the constraints regarding complexity, syntax, IDE support and infrastructure are very different from those targeted at non-programmers, we do not compare further in related work.

10 Conclusions

Summary We have described the development of a domain-specific language for payroll applications at DATEV. The DSL reduces complexity in terms of the core domain and infrastructure dependencies (RQ1), increases quality by simplifying testing, immediate feedback, and a step-wise build process (RQ2), is accessible to non-expert programmer end users (RQ3) and integrates with existing architectures and build pipelines, while keeping deployment options flexible (RQ4). This way, the language helps DATEV address core business challenges including keeping track with evolving law (C1), the need to develop new and innovative products faster (C2) and running those on a wide variety of platforms (C3). While not everything was smooth sailing, the DSL is now in productive use.

Conclusions At a more general level, all involved parties agree that the goals set out for the DSL-based development process have largely been reached, as far as we can tell after a few years of development and use. Both the business programmers and the infrastructure developers recognize the value of separating the domain and technical concerns. Business programmers have been overheard saying that "we really don't care about the technical implementation in the data center" are happy that they can implement, test and deploy (!) new functionality reliably within a single sprint, something that was not feasible before. The infrastructure developers are also happy because they "don't have to care about the complexity of payroll calculation", and that they are able to ship cross-cutting optimizations in the generated code with relative ease. The pain-free migration from JEE to Spring also drove home this benefit.

The Future Based on the positive experience with the DSL, it will be expanded in the future. As of now, two major extension of the scope of the DSL are currently being adressed. The first is data transfer to external consumers (such as government agencies) where we are working on language features to specify the respective data mappings. The second extension is the creation of reports and lists, where the DSL will support query and aggregation of data.

Acknowledgements

We thank Bastian Kruck and the reviewers of the book for detailed and very useful feedback on this paper. Our thanks also goes to the anonymous reviewers of the MPS book.

References

1. B. T. M. Anh, S. Stinckwich, M. Ziane, B. Roche, and H. T. Vinh. Kendrick: A domain specific language and platform for mathematical epidemiological modelling. In *The 2015 IEEE RIVF International Conference on Computing & Communication Technologies-Research, Innovation, and Vision for Future (RIVF)*, pages 132–137. IEEE, 2015.
2. T. Antao, I. M. Hastings, and P. McBurney. Ronald: A domain-specific language to study the interactions between malaria infections and drug treatments. In

BIOCOMP, pages 747–752, 2008.
3. C. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.
4. J. Barzdins, K. Cerans, A. Kalnins, M. Grasmanis, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, and A. Zarins. Domain specific languages for business process management: a case study. In *Proceedings of DSM*, volume 9, pages 34–40, 2009.
5. M. Broy, S. Kirstan, H. Krcmar, B. Schätz, and J. Zimmermann. What is the benefit of a model-based design of embedded software systems in the car industry? *Software Design and Development: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications*, page 310, 2013.
6. T. Bruckhaus, N. Madhavii, I. Janssen, and J. Henshaw. The impact of tools on software productivity. *IEEE Software*, 13(5):29–38, 1996.
7. M. Chinosi and A. Trombetta. BPMN: An introduction to the standard. *Computer Standards & Interfaces*, 34(1):124–134, 2012.
8. D. R. Christiansen, K. Grue, H. Niss, P. Sestoft, and K. S. Sigtryggsson. An actuarial programming language for life insurance and pensions. In *Proceedings of 30th International Congress of Actuaries.*, 2013.
9. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.
10. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992.
11. L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.
12. S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, page 7. ACM, 2012.
13. S. Erdweg, T. Van Der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, pages 197–217. Springer, 2013.
14. E. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.
15. M. Faerber, S. Jablonski, and T. Schneider. A comprehensive modeling language for clinical processes. In *ECEH*, pages 77–88. Citeseer, 2007.
16. W. Feurzeig and G. Lukas. LOGO - a programming language for teaching mathematics. *Educational Technology*, 12(3):39–46, 1972.
17. S. P. Florence, B. Fetscher, M. Flatt, W. H. Temps, T. Kiguradze, D. P. West, C. Niznik, P. R. Yarnold, R. B. Findler, and S. M. Belknap. POP-PL: a patient-oriented prescription programming language. In *ACM SIGPLAN Notices*, volume 51, pages 131–140. ACM, 2015.
18. F. Hermans, M. Pinzger, and A. Van Deursen. Domain-specific languages in practice: A user study on the success factors. In *International Conference on Model Driven Engineering Languages and Systems*, pages 423–437. Springer, 2009.
19. G. Huber and R. Wälzlein. Problem und lösung beschreiben. *Java Magazin*, 2017(1), 2017.
20. T. Johnston. *Bitemporal data: theory and practice*. Newnes, 2014.
21. E. Jürgens and M. Feilkas. Domain specific languages. 2006.
22. J. Kärnä, J.-P. Tolvanen, and S. Kelly. Evaluating the use of domain-specific modeling in practice. In *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*, 2009.

23. R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th international conference on Software engineering*, pages 542–552. IEEE Computer Society, 1996.
24. H. Kolk and M. Voelter. Democratizing software creation. `http://voelter.de/data/presentations/KolkVoelter_IntentionalSoftware.pdf`, 2008. Presentation at OOP 2008 conference.
25. G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson. Assessing the state-of-practice of model-based engineering in the embedded systems domain. In *International Conference on Model Driven Engineering Languages and Systems*, pages 166–182. Springer, 2014.
26. P. Liggesmeyer and M. Trapp. Trends in embedded software engineering. *IEEE software*, 26(3), 2009.
27. J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16, 2010.
28. S. McDirmid. Usable live programming. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 53–62. ACM, 2013.
29. A. Mediratta. *A generic domain specific language for financial contracts*. PhD thesis, Rutgers University-Graduate School-New Brunswick, 2007.
30. Metacase Consulting. DSM examples. `https://metacase.com/cases/dsm_examples.html`, 2019.
31. Ming-Yee Iu. Programming basics: A website teaching people how to program. `http://www.programmingbasics.org/en/`, 2018.
32. Object Management Group. Decision model and notation specification version 1.2. `https://www.omg.org/spec/DMN`, 2019.
33. G. J. Pace and M. Rosner. A controlled language for the specification of contracts. In *International Workshop on Controlled Natural Language*, pages 226–245. Springer, 2009.
34. B. Pepels and G. V. v. Zanten. Model driven software engineering in the large: Experiences at the dutch tax and customs service (industry talk). In *Proceedings of the 1st Industry Track on Software Language Engineering*, ITSLE 2016, pages 2–2, New York, NY, USA, 2016. ACM.
35. S. Peyton Jones, J.-M. Eber, and J. Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *ACM SIGPLAN Notices*, volume 35, pages 280–292. ACM, 2000.
36. C. Simonyi, M. Christerson, and S. Clifford. Intentional Software. In *ACM SIGPLAN Notices*, volume 41, pages 451–464. ACM, 2006.
37. J. Stenberg. Domain-driven design with onion architecture. `https://www.infoq.com/news/2014/10/ddd-onion-architecture`, 2014.
38. J. Stoel, T. v. d. Storm, J. Vinju, and J. Bosman. Solving the bank with rebel: on the design of the rebel specification language and its application inside a bank. In *Proceedings of the 1st Industry Track on Software Language Engineering*, pages 13–20. ACM, 2016.
39. A. van Deursen. Domain-specific languages versus object-oriented frameworks: A financial engineering case study. *Smalltalk and Java in Industry and Academia, STJA 97*, pages 35–39, 1997.
40. A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.
41. M. Voelter. Language and IDE development, modularization and composition with MPS. In *GTTSE 2011*, LNCS. Springer, 2011.
42. M. Voelter. The design, evolution, and use of KernelF. In *International Conference on Theory and Practice of Model Transformations*, pages 3–55. Springer, 2018.

43. M. Voelter. Fusing modeling and programming into language-oriented programming. In *International Symposium on Leveraging Applications of Formal Methods*, pages 309–339. Springer, 2018.
44. M. Voelter. Programming basics: How to think like a programmer. `https://markusvoelter.github.io/ProgrammingBasics/`, 2018.
45. M. Voelter. The hidden layer between the fachlichkeit and the -ilities. `https://medium.com/@markusvoelter/the-hidden-layer-between-the-fachlichkeit-and-the-ilities-7d850fde00bf`, 2019.
46. M. Voelter, B. Kolb, K. Birken, F. Tomassetti, P. Alff, L. Wiart, A. Wortmann, and A. Nordmann. Using language workbenches and domain-specific languages for safety-critical software development. *Software & Systems Modeling*, pages 1–24, 2018.
47. M. Voelter, B. Kolb, T. Szabó, D. Ratiu, and A. van Deursen. Lessons learned from developing mbeddr: a case study in language engineering with MPS. *Software & Systems Modeling*, Jan 2017.
48. M. Voelter, T. Szabó, S. Lisson, B. Kolb, S. Erdweg, and T. Berger. Efficient development of consistent projectional editors using grammar cells. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 28–40. ACM, 2016.
49. M. Voelter, A. van Deursen, B. Kolb, and S. Eberle. Using c language extensions for developing embedded software: A case study. In *OOPSLA 2015*, 2015.
50. D. Weintrop, A. Afzal, J. Salac, P. Francis, B. Li, D. C. Shepherd, and D. Franklin. Evaluating coblox: A comparative study of robotics programming environments for adult novices. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, page 366. ACM, 2018.
51. J. M. Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, 2006.