Migrating Insurance Calculation Rule Descriptions from Word to MPS

Niko Stotz and Klaus Birken

Abstract Zurich Insurance used to specify their calculation rules in form-based prose and pseudo-code, which was subsequently implemented by an external party. The resulting long turnaround time seriously affected Zurich's time-to-market. Zurich and itemis replaced these specifications with the *FuMo* DSL. Its productive usage has been ongoing for more than 2 years now.

Due to MPS' projectional editor, the DSL closely resembles both the previous forms and the well-known pseudo-code. The language's generator removed the external party from the development round-trip. Consequently, the turnaround time went down by several orders of magnitude.

This project imported existing calculation rules from their C implementation and lifted them to FuMo DSL. We hid the complexity of C while lifting, so the end-users can focus on domain aspects. MPS' language integration enabled a clean design of the FuMo DSL, while edge cases could be handled with special concepts or in embedded C blocks.

We split the import process into small steps that could be validated independently. We assured all steps could be executed by the development team. By annotating the execution with trace logging and comparing the trace logs of the original source code with the generated one, we could handle large batches of similar import issues efficiently.

N. Stotz (✉)
Canon Production Printing Netherlands B.V., Venlo, The Netherlands
e-mail: niko.stotz@nikostotz.de

K. Birken
itemis AG, Stuttgart, Germany
e-mail: klaus.birken@itemis.de

A. Bucchiarone et al. (eds.), *Domain-Specific Languages in Practice*,
https://doi.org/10.1007/978-3-030-73758-0_6

1 Introduction

This chapter reports on a project at Zurich's [11] life insurance branch in Germany. Zurich and itemis [3] started this project to shorten time-to-market by improving the product implementation process. We achieved this goal by migrating technical product descriptions from Microsoft Word to an MPS-based domain-specific language (DSL). To ease adoption, we leveraged MPS' projectional editor by rendering the DSL very similar to the original Word documents. Originally, the Word documents were implemented in C by an external service provider. We superseded this process step by a generator from the DSL to C code, cutting down the turnaround time for each change from days or weeks to seconds.

Info
This chapter focuses on the implementation side of Zurich's project with itemis. Organizational aspects are out of scope; domain-specific details are only mentioned insofar as they affect technical decisions.

Section 2 introduces Zurich, key aspects of the insurance industry, and the project's context. It provides an overview of the way of working in Zurich's relevant departments prior to this project, describes participants and artifacts, and explains the challenges Zurich was facing. The participants include the *IT department* to translate requirements to technical descriptions and an *external service provider* for implementation. They worked with artifacts for data model definition, functional specification, implementation in C, and functional and regression tests.

Section 3 points out how this project met its challenges. These challenges can be categorized as follows: (1) tooling challenges, including ad hoc version control, unstructured pseudo-code in Word-based technical product descriptions, and inaccurate searches; (2) implementation challenges, like memory allocation based on different libraries, inconsistencies between technical product descriptions and implementation, and coarse test result granularity; (3) process challenges, including required C language knowledge for nonprogrammers, duplicated implementation effort, test execution only available on mainframe systems, high communication effort, and long turnaround time. This section furthermore motivates technological choices and design decisions, including MPS and key language characteristics.

Section 4 describes the project's results and delves into issues during its implementation. The section discusses MPS' benefits and shortcomings for such a migration project. It describes in detail both our first—ultimately failing—attempt to import the existing code base and the key differences that turned the second attempt into a success.

Section 5 summarizes the advantages and disadvantages of the MPS platform for this project and evaluates the project's results.

2 Project Description and Challenges

Zurich is an international insurance company offering a wide range of insurance products. An *insurance product* is one specific insurance a customer can buy. One example might be *term life insurance with constant sum insured*. They continuously develop new products or evolve existing products, for reasons such as changing regulatory requirements, technical innovation, and business innovation. These products are core assets to every insurance company. As such, they need to be maintained on the highest level of quality.

The industry faces specific challenges for their products. Insurances operate in a mature market, so there is always an alternative insurance provider with a similar offering. With such fierce competition, time-to-market is a critical success factor.

Especially for life insurance, single insurance policies might run for a long time (i.e., decades): think of a term life insurance taken out by a young mother in her 20s. Old policies must keep working as they were agreed upon, while new policies of the same product must adhere to changed laws or updated risk assessments. Thus, the IT systems must support different versions of the same product in parallel.

Prior to this project, the development process for a new or changed product consisted of these high-level steps (see Fig. 1):

1. *Product modelers* specified all domain-related details. They are experts on insurance maths and devised all calculation rules. The results are documented in a Microsoft Word *specification* document.
2. The *IT department* analyzed these documents and derived technical descriptions called *FuMo*s (*Funktionenmodelle*, function models). FuMos are rigidly structured Microsoft Word documents, written in German. Conceptually, a FuMo resembles a programming language function. In the *Versicherungsanwendungsdatenmodell* (insurance application data model, VADM), the IT department described the data structures shared among all FuMos.

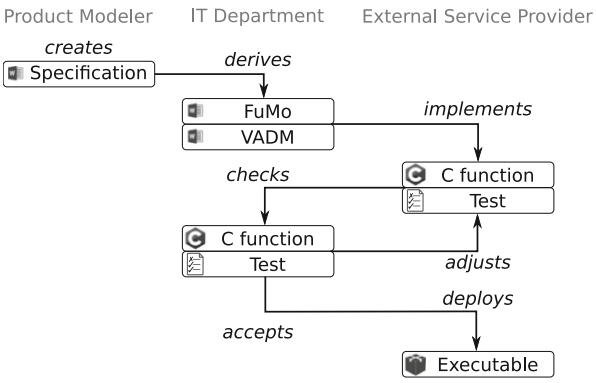


Fig. 1 Flow diagram of the original product development process before the start of this project

3. An *external service provider* received the FuMos, implemented them in C code, and wrote tests.
4. The IT department checked the implementation. In case of issues, the IT department advised the external service provider; they adjusted the implementation. These steps were repeated until all issues were resolved. Each round-trip took days to weeks—an obvious bottleneck for swift development.
5. The external service provider moved the changes into production.

Info
The remainder of this section discusses the relevant artifacts (e.g., VADM, FuMo, C sources) and participants of the product development process in detail.

2.1 Data Model Definition with VADM

The *Versicherungsanwendungsdatenmodell* (insurance application data model) is a global data structure shared among all FuMos. It was written manually in Word in tabular form, one row per element. It contains all parameters referred to by FuMos (see Fig. 2). Each parameter is assigned to a top-level structure (*Struktur vertrag*, structure contract). Next to columns *Bezeichnung* (description) to denote the nesting level and technical name, and column *Typ* (type) to specify the C data type, the table contains proprietary Zurich information not relevant to this chapter. References to other parts of VADM are denoted by an asterisk in the description column. Some references are kept for technical reasons, especially performance optimization (e.g., `2 *hmr_ptr`). Note the strong technical influence on this domain-level artifact—none of C data types, pointer notation, or performance optimizations are domain specific. Composite attributes use bold formatting and only provide a description and attribute of VADM. Attribute names are unique within their parent composite structure, but not globally. Composite structures could be embedded in more than one parent composite structure, or—with different names—more than once in the same parent composite structure.

VADM also provides static input data such as mortality tables (not depicted). This data was written in Word in the obvious table form: one table per structure, one column per attribute, one row per entry.

Bezeichnung	Typ	Attribut des VADM	Entität des VADM	Proj.Code
1 vertrag		**Vertragsdaten**		
2 pd_id	char[]	Kenn-Produkt	LV-VERTRAG	
2 lvbegt	Long	Beginn-Termin-technisch	LV-VERTRAG	
2 ruewbeg	Long	Rueckwirkender-Beginntermin	LV-VERTRAG	
[illegible]	[illegible]	Beginn-Termin-Sta[illegible]	[illegible]	[illegible]
2 kz_bfr	short	Kz-f-Beitragsfreistellung	LV-VERTRAG	CH0007.53
2 kz_bfr_mahnl	short	Kz-f-Beitragsfreistellung-Mahnlauf	LV-VERTRAG	CH0010.12
2 bfr_termin_stufe	long	BFR-Termin-Stufe	LV-VERTRAG	D0023.34
2 kz_rkw_gar	Short	Kz-f-RKW-Garantiert	LV-VERTRAG	CH0007.70
2 alpha_z	double	Abschlusskostensatz-Zulagen	LV-VERTRAG	D0014.08B
2 *hmr_ptr	Void	techn. Verweis auf produkttypspez. Hoch- und Modellrechnungen	UNTYP. POINTER	
2 *bvd_ptyp_ptr	Void	techn. Verweis auf produkttypspez. Beschreibende Vertragsdaten	UNTYP. POINTER	
2 *hochrechnung_ad_ptr	Void	techn. Verweis auf Hochrechnungs-Struktur AD	UNTYP. POINTER	
2 *vk_liste	VK	techn. Verweis auf VK	POINTER	
2 vbtg		**Gesamtbeitrag**		
3 zw	Short	Kenn-Zahlweise	INKASSO-ZAHLWEISE-VEREINBARUNG	
3 zw_anf	Short	Kenn-Zahlweise-anfaenglich	INKASSO-ZAHLWEISE-VEREINBARUNG	VVG
3 vzb	Double	Zahlbeitrag-V-ratierlich	LV-VERTRAG	
3 vzb_vst	Double	Versicherungssteuer-Zahlbeitrag-V	LV-VERTRAG	
3 btrabgl	Double	Beitragsabgleich	LV-VERTRAG	
3 btrabgl_rea	Double	Beitragsabgleich-f-Reaktivierung	LV-VERTRAG	

© Release 18.1 gespeichert: 14.10.2015 3-4

Fig. 2 Example of a Word VADM (*Versicherungsanwendungsdatenmodell*, insurance application data model). It describes the global data structure all code is operating on. Only the first three columns are relevant to this chapter[1]

Info
In this chapter, we use the term *attribute* for one VADM member. The term *parameter* is used when a FuMo uses a VADM attribute to exchange data with the FuMo's caller. This reflects Zurich's terminology.

[1]Figures 2, 3, 5, 6, and 10 depict original artifacts from the customer's business domain and the corresponding DSLs. This encompasses not only the content but also the German language. In order to convey the domain as original as possible, we decided against translating the artifacts and keep them in German.

2.2 Functional Specification in FuMo

A FuMo (*Funktionenmodell*, function model) describes the technical implementation of domain functionality. From a programmer's point of view, a FuMo is a function with side effects.

2.2.1 Description

The roughly 1000 FuMos were written manually in Word, following a rigid structure (compare to annotations in Fig. 3):

1. Technical meta-information, like function name (*Funktion*), containing C source file (*Programmquelle*), VADM attributes used as parameters (*verwendete Attribute*), and other FuMos used by this one (*aufgerufene Funktionen*)
2. Domain meta-information, like product type (*Produkt-Typ*) and product category type (*PK-Typ*)
3. Administrative meta-information, like current version (*Status*) and version history (table)
4. Functionality described in German prose text (*Verarbeitungen*), interspersed with references to VADM attributes (light gray rectangle) and math formulas (medium gray rectangle)
5. Functionality described in pseudo-code (remainder of document), containing references to VADM attributes (bold) and calls to other FuMos (italics, dark gray rectangle)

The function name and source file were used by the external service provider's implementation. The referenced VADM attributes and other FuMos were maintained manually. The file's version history was kept inside the document. During active development of a FuMo, Word's *track changes* functionality was used regularly. All descriptions and names were written in German language. Inside the prose text and the pseudo-code, formatting was used to refer to parameters or used FuMos.

The pseudo-code leveraged the usual concepts of a structured, procedural programming language. It was not specified formally or informally, but used by *common sense*. Blocks were denoted by indentation and/or block end statements (`Falls kz_rw_param = 12 ... sonst ... Ende Falls kz_rzw_param = 12`, *if*... *else*... *end* block). References to VADM attributes were implicit, i.e., only distinguishable from local variables by formatting—if applied correctly. VADM attribute references were implicitly scoped: whether the attribute was a member of a composite type, and of which instance of the composite type, was inferred from the context. Think of a nested dot notation with only the rightmost part actually written. Parameter passing semantics (pass-by-value vs. pass-by-reference) was implicit.

Funktionenmodell

Formale Beschreibung

a	**Funktion:**	berbwvekFF		
a	**Programmquelle:**	vmsctfa1.c		
b	**Produkt-Typ:**	FONDS	**PK-Typ:**	KAPITAL-KONTO
		Name	Verw.	Entität
a	**verwendete Attribute:**	lkm_akt_param	E	PARAMETER
		lkm_faell_param	E	PARAMETER
		ber_zweck_param	E	PARAMETER
		kz_rzw_param	E	PARAMETER
		bwvek	A	RETURN
a	**aufgerufene Funktionen:**	berbweinzelFF		
		VTRKermbtgfaellFF		
c	**Status:**	18.1		

c zuletzt geändert von ... am ...	Joachim Kaiser, 09.10.2014

d **Verarbeitungen**

Die Funktion liefert den Barwert per lkm_akt_param des vorschüssigen Zahlungsstroms der Höhe 1 von Monat lkm_akt_param bis lkm_faell_param – jeweils einschließlich. Zahlungszeitpunkte sind jeweils die Monatsbeginne, also **lkm_akt_param** -1 bis **lkm_faell_param** – 1.

Der Parameter **kz_rzw_param** steuert die zu berücksichtigende Zahlweise des Zahlungsstroms. Möglich sind zur Zeit nur die Ausprägungen 0 (Zahlungen zu den Beitragsfälligkeiten) und 12 (monatliche Zahlungsweise).

e Schleife über lkm_faell_hilf = **lkm_akt_param** bis **lkm_faell_param**

Falls **kz_rzw_param** = 12

kz_bf_hilf = 1

sonst

kz_bf_hilf = *VTRKermbtgfaellFF*(lkm_faell_hilf)

Ende Falls **kz_rzw_param** = 12

bwvek = **bwvek**

+ kz_bf_hilf * *berbweinzelFF*(**lkm_akt_param**, lkm_faell_hilf - 1, **ber_zweck_param**)

Ende Schleife

return **bwvek**

berbwvekFF.docx 14.10.15 19:14 Seite 1/1

Fig. 3 Example of a Word FuMo (*Funktionenmodell*, function model). It describes the technical implementation of domain functionality. The annotations correspond to Fig. 6[1]

In theory, the prose text and pseudo-code would be equivalent and geared towards different audiences: prose text to explain the FuMo to domain experts, and pseudo-code to enable programmers to implement the FuMo. In most cases, this pseudo-code was detailed enough to be executed by a computer.

2.2.2 Challenges

Version Control The FuMo and VADM documents were kept on a shared network drive without formal version control, leading to well-known issues: any cooperative or conflicting change to a document had to be coordinated manually; keeping a change log was up to each developer's discipline.

FuMo Structure FuMos were written manually in Word, i.e., without any formal consistency check. The pseudo-code was effectively a programming language, but written without any programming tools: neither syntax check nor named reference resolution or type compatibility check was available to the authors.

Without syntax check, block boundaries might be ambiguous. FuMos used both indentation and block end statements to denote block boundaries. They might be inconsistent or ambiguous (e.g., "Does this statement belong to the `if` statement or the outer loop?").

As references had been updated manually, inconsistencies were inevitable: the lists of attributes and used FuMos had not been updated to changes in the functional description, and name changes had not been propagated to all references. The formatting of references to VADM attributes and used FuMos had not been applied in all cases. To start a new FuMo, the developer had copied a similar existing one. This practice had increased the likelihood of outdated references.

Implicit scoping of VADM attribute references was prone to omitted intermediate steps in the attribute access chain. Think of a nested dot notation with missing segments in between. As VADM attribute names are not globally unique, references could be ambiguous.

Due to the missing language specification, different authors used different wording to describe the same concept, e.g., both *Schleife über x = … bis …* (loop over x = … until …) and *Schleife von x = … zu …* (loop from x = … to …). The lack of specification leads to undefined semantic details: in the above example, will the upper limit value be contained within the range of the loop?

2.3 Executable Implementation in C

FuMos and VADM structures were implemented in the C programming language.

2.3.1 Description

The C implementation consisted of about 200,000 lines of code. It was compiled for verification with Microsoft Visual Studio by the IT department and for production on a mainframe with the system's own compiler. In most cases, one FuMo was implemented by one C function; it was up to the implementer to split the implementation in more functions, if necessary. VADM composite attributes mapped to C `structs`, with simple VADM attributes as members. The implementation leveraged an extensive library of insurance-specific functions, like depreciation or life expectancy.

The source code files were stored on a shared network drive, similar to FuMos. A basic version control scheme kept old versions of C source files by renaming the old version of the file and appending a version number. The version number loosely related to the *Status* field in FuMo.

The code base evolved alongside the executing system. Thus, a lot of different engineers worked on the code over time, leading to inconsistent style and patterns. Most notably, the code base contained three different APIs for memory allocation.

The run-time environment initialized one complete VADM structure. FuMos had access to this complete structure, read and/or wrote parts of the structure, and returned to the caller. The caller implicitly knew which attribute would contain the expected results. Other attributes would be used to store intermediate results or to transfer intermediate results between multiple FuMos participating in the same processing chain. Value function parameters rarely passed domain-related information; mostly, they were used to provide temporary values like counters or array indices.

2.3.2 Challenges

Version Control The basic version control scheme led to similar issues as mentioned in Sect. 2.2.2: it requires lots of manual coordination and complicates tracking changes.

Memory Allocation During this project, we had to unify the memory allocation to one API. This process revealed typical issues related to this topic like use-after-free, duplicate memory usage, or mismatches between the APIs used to allocate and free the same chunk of memory.

Global VADM Structure One global structure for passing around input and output parameters removes all locality from single functions; thus, the system effectively has only one global scope. Two functions can only work in the same context if they *happen* not to use the same VADM attributes. Without a formal declaration which VADM attributes are used by each function for input, output, or temporary storage, there is no way to assure correct results besides individual inspection.

2.4 Functional and Regression Tests

The calculation rules implemented by FuMos constituted core business value for Zurich. Appropriately, extensive functional and regression tests were in place.

2.4.1 Description

The test data accumulated to about 1.5 terabytes; a complete test run on a mainframe took several days. The test setup integrated tightly with VADM processing. Each test loaded a specific state of the VADM structures, ran the FuMo under test, and binary compared the VADM structures afterwards with the expected result. Members of the IT department created the test data during development.

2.4.2 Challenges

Execution Environment As both test setup and expected result consisted of all values of the complete VADM structure, they were tightly coupled to the mainframe execution environment. They could not be executed on a PC. Thus, the mainframe staging system that ran tests became a bottleneck.

Test Result Granularity The test output was binary compared with the expected result. This way, we could only determine automatically whether a test succeeded or failed; we could not provide a reason for failure. Any failing test required detailed analysis of the test output data and thus considerable manual effort.

2.5 IT Department Team Authoring VADM and FuMo

The members of the involved team in the IT department possessed thorough insurance domain knowledge, gained through years of experience. They leveraged their expertise to translate insurance specifics from the specification documents into technical FuMo documents. The IT department also verified the C code created by the external service provider by means of manual code inspection.

2.5.1 Description

The team consists of five engineers and one team lead. One team member had a formal computer science background. Their prior experience with version control was limited. None of them had worked with modeling tools before.

IT department staff regularly searched through all existing functionality. Rationales to do so included finding all contexts a FuMo is used in or assuring the interpretation of an attribute's value.

The IT department had to assure the correct implementation of FuMos by the external service provider. Thus, the team members needed to be proficient in understanding C source code. Due to the domain's complexity, simply reading the C source code was not sufficient; they also needed to execute, debug, and experiment with the C implementation.

The team members sometimes prototyped parts of a FuMo for validation in C, before sending it to the external service provider.

2.5.2 Challenges

Search Through C Code Base The staff could not use their primary artifacts, the Word files containing VADM and FuMos, for global searches. Word's search capability had proven to be insufficient in both flexibility and performance. Thus, the team members regularly used the C implementations as search corpus. Only the C implementations guaranteed correct named references to parameters, functions, etc. Correct names were crucial, as most searches looked for function or parameters usage—similar to *find all references* functionality in an *integrated development environment* (IDE).

C Language Knowledge For all of searching, prototyping, and assuring the correct implementation, team members had to know C. Thus, each member of the team had to understand both the domain (to interpret the specification documents) and the intricacies of the C language. These diverse requirements seriously limited staffing options.

Duplicated Implementation Effort As the only available executable environment was C code, team members could only validate a FuMo via a C prototype implementation. However, implementing C should have been the responsibility of the external service provider. Thus, team members had to delve into tasks that were more suitable to other parties, and the external service provider had to spent effort into reworking or re-implementing the prototype.

2.6 External Service Provider Implementing Executable C and Tests

The external service provider accounted for all implementation, testing, performance, monitoring, scalability, and security.

2.6.1 Description

Only the external service provider's engineers were allowed to change the C implementation or tests. The majority of the involved engineers were located overseas and did not speak German as their native language.

2.6.2 Challenges

Inconsistencies Between FuMo and C Implementation A FuMo and the corresponding C implementation were maintained by different parties. Also, they were updated without any strong feedback mechanism or notification, especially during iterative issue resolution. Thus, they often grew apart.

Two different FuMos, or even two parts of the same FuMo, might have been implemented by different software engineers. Even though the FuMo used the same pseudo-code, it might have been translated to C in different ways.

High Communication Effort The IT department and the external service provider were from different corporations, native languages, and professional backgrounds. This setup increased communication overhead, posed the chance of misunderstandings, and potentially added commercial aspects to the discussion.

Long Turnaround Time The IT department assured correct functionality of the implementation, but only the external service provider could apply changes. Any time IT department members spotted incorrect functionality, they had to describe the issue, hand it over to the external service provider, wait for the implementation, and could verify the change only then. The round-trip could cause the IT department members wait for a long time (days to weeks) until they would know the definitive result of any change.

3 Proposed Solution

We proposed to migrate VADM and all FuMos to MPS models and generate the C implementation. The revised product development process is depicted in Fig. 4. It removed the *implement–check–adjust* cycle between the IT department and the external service provider.[2] The test creation process remained the same. The IT department continued to define FuMos and VADM, but in MPS instead of Word. The MPS models were stored in the *Subversion* [1, 7] version control system. All non-domain aspects were moved to support libraries. This approach removed most of the identified shortcomings.

[2]Close cooperation is still required to ensure nonfunctional requirements and infrastructure enhancements.

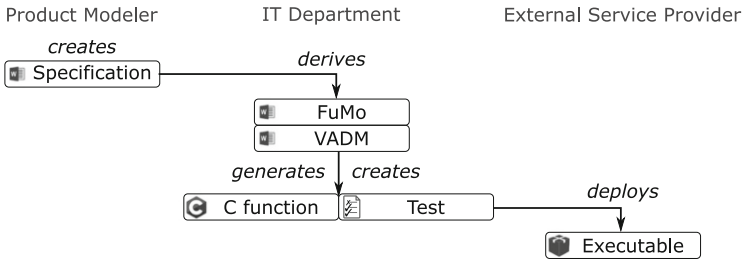


Fig. 4 Flow diagram of the revised product development process, as resulted from this project

Version Control FuMo models are versioned like any other development artifact. The versioning system provides proven diff/merge support, completely integrated into MPS. C implementations do not need to be versioned any more, as they can be regenerated at any time.

FuMo Structure By formalizing the FuMo DSL, we provide all the regular IDE tooling: entering invalid or inconsistent syntax becomes impossible by design, auto-completion supports the user with available choices, named references are replaced by technical references and guarantee consistency, and model validation provides instant feedback. We have implemented a generator, so inherently each construct has defined semantics. Of course, all technical meta-information listed in the FuMo header is implicit: we automatically collect the lists of parameters and used FuMos from the FuMo DSL.

Memory Allocation The FuMo DSL does not provide direct access to memory management; it is handled by the generator. We use the same memory allocation API throughout the generator.

Global VADM Structure We do not change this fundamental software design choice. For once, the approach has been chosen for a good reason: it minimizes data copying and enables high performance. Also, such a change has been out of the scope of the project, and it would not have been wise to combine such a fundamental implementation change with the technology migration at hand. However, due to the FuMo DSL, we know exactly which FuMo (or function) accesses which VADM attribute. This offers a sound basis for future changes, if desired.

Test Execution Environment Although not initially planned, we had to enable test execution on PCs to finish the migration to MPS (see Sect. 4.4). Based on this work, Zurich now runs tests in an automated nightly build.

Test Result Granularity For our migration efforts, we semiautomated more granular failure reports through trace logging (see Sect. 4.4). However, this required invasive changes to the production code and would hamper performance seriously. Thus, these changes were only temporary.

Enable Accurate Searches Most searches look up named references. By providing first-class technical references, we replaced these searches by straightforward linking and *find all references* commands. FuMo models become the definitive source, and generation guarantees consistency of FuMo and C implementation. Thus, we cannot miss any reference from the model.

C Language Knowledge and Duplicated Implementation Effort IT department staff writes FuMos in FuMo DSL and can immediately generate, compile, and execute the resulting implementation—prototyping and implementing a FuMo uses the same tool. This means searching and prototyping does not require detailed knowledge of the C language any more. Programming skills are still required for debugging purposes.

Inconsistencies Between FuMo and C Implementation All C implementations are generated from the FuMo models. Therefore, we cannot have any inconsistencies. As we use the same generator for all FuMos, all C implementations for one FuMo DSL concept must be identical.

High Communication Effort With code generation, writing FuMos and implementing them in C happen at one place by one party, thus removing any communication overhead. The external service provider stays accountable for nonfunctional aspects of the system. They can focus on support libraries and operational aspects. Of course, both parties need to stay in close alignment.

Long Turnaround Time Code generation delivers the C implementation of every FuMo within seconds, i.e., several orders of magnitude faster than the previous process. This provides the IT department with immediate feedback and direct control of the outcome.

3.1 Solution Technologies

We proposed MPS as implementation technology for several reasons. To ease adoption, we kept a form-like user experience similar to the existing FuMos. This is easily feasible with projectional editors. Similarly, projectional editors support VADM's tabular style. Intermixing prose text in FuMos with formal math expressions and links to parameters would be hard or even impossible with parser-based systems.

MPS' language extension mechanism supports clean language design decisions while providing an escape mechanism for edge cases in legacy code. We could design the FuMo DSL on its existing level of abstraction. Edge cases like irregular pointer access were handled by language concepts unavailable to the end-user, while performance optimizations like pointer arithmetic could be represented by embedded C code.

itemis knows MPS very well, rendering this technology the obvious choice. We could leverage our experience with *mbeddr* [2, 4, 10] as MPS-based C implemen-

tation and generator to C source code. Our internal importer from C source code to mbeddr C models enabled the import of existing FuMo implementations. Zurich tasked itemis with maintenance, migration, and further development, relieving them from MPS development.

We proposed Subversion as a version control system, as it was available within Zurich. It is less complex than git, lowering the initial threshold especially for users unfamiliar with version control systems. MPS supports Subversion out of the box.

3.2 VADM

The Word variant of VADM was represented as a set of tables with one row per attribute (see Sect. 2.1). Static input data like mortality tables was also maintained as Word documents containing tables.

3.2.1 Language Design

Our MPS language followed the same table-based approach for both parameters and input data (see Fig. 5). Optionally, we showed known references to this attribute from both VADM and FuMo (not depicted).

3.2.2 Language Implementation

The VADM language is straightforward. The concepts were translated one-to-one from the semantics of rows and columns in Word, with *slisson tables*[3] we could implement the editors easily.

At first, we tried to keep exactly the same kind of granularity as in Word, i.e., one root node for each Word document. Some of these root nodes became too large to render performantly in MPS. We split them up at the first nesting level. Due to automatic references, the additional effort for the end-user was negligible: They can follow references similar to hyperlinks or *find all references* to navigate in the opposite direction.

We tried to simplify this kind of navigation even further by listing all usages of each VADM attribute in an additional column. However, this means running *find all references* for each attribute. This only performs well for small VADM documents or not heavily used attributes. Thus, we hid this column by default.

3.3 FuMo

The visual design of our language tried to follow the Word representation closely (see Fig. 6). We used the same layout for domain and technical meta-information.

[3]Editor extension for advanced table structures [5].

VADM-Hauptstruktur rg_kk

	Bezeichnung	Typ	Attribut des VADM	Entität des VADM	Proj.Code
1	rg_kk		nur zur Übung		
2	fo_[blurred]_[blurred]tz	Ganzzahl	Attribut-Langname	<no entity>	<no project>
2	[blurred]_[blurred]tz	Kommazahl	Attribut-Langname	<no entity>	<no project>
2	ko_ra_id	Zeichenkette	Attribut-Langname	<no entity>	<no project>
2	kz_zus_gar	Bool	Attribut-Langname	<no entity>	<no project>
2	zm	Ganzzahl	Attribut-Langname	<no entity>	<no project>
2	[blurred]_[blurred]_akt	Kommazahl	Attribut-Langname	<no entity>	<no project>
2	zw	Kommazahl	Attribut-Langname	<no entity>	<no project>
2	vtrk_zb	Ganzzahl	Attribut-Langname	<no entity>	<no project>
2	kz_mandant	Zeichenkette	Attribut-Langname	<no entity>	<no project>

Fig. 5 Example of an MPS VADM (partially blurred for confidentiality)[1]

Funktionenmodell berbwvekFF

Formale Beschreibung

Funktion: berbwvekFF
Programmquelle: vmsctfa1.c
Produkt-Typ: Fonds **PK-Typ:** Kapital-Konto
Status: 18.1

Parameter-Attribute
lkm_akt_param
lkm_faell_param
ber_zweck_param
kz_rzw_param

Verwendete VADM-Attribute
Keine verwendeten VADM-Attribute, werden automatisch hinzugefügt

Rückgabe-Attribut
bwvek

aufgerufene Funktionen
VTRKermbtgfaellFF (a)
berbweinzelFF (a; b; c)

Beschreibung
Die Funktion liefert den Barwert per @lkm_akt_param des vorschüssigen Zahlungsstroms der Höhe 1 von Monat @lkm_akt_param bis @lkm_faell_param – jeweils einschließlich. Zahlungszeitpunkte sind jeweils die Monatsbeginne, also #lkm_akt_param – 1# bis #lkm_faell_param – 1#. Der Parameter @kz_rzw_param steuert die zu berücksichtigende Zahlweise des Zahlungsstroms. Möglich sind zur Zeit nur die Ausprägungen 0 (Zahlungen zu den Beitragsfälligkeiten) und 12 (monatliche Zahlungsweise).

Hilfsvariablen
kz_bf_hilf

Verarbeitungen
```
Schleife über lkm_faell_hilf = lkm_akt_param bis lkm_faell_param
  Falls kz_rzw_param = 12
      kz_bf_hilf = 1
    sonst
      kz_bf_hilf = VTRKermbtgfaellFF (lkm_faell_hilf)
  Ende Falls kz_rzw_param = 12
  bwvek = bwvek + kz_bf_hilf * berbweinzelFF (lkm_akt_param; lkm_faell_hilf – 1; ber_zweck_param)
Ende Schleife über lkm_akt_param bis lkm_faell_param

return bwvek
```

Fig. 6 Example of an MPS FuMo (the annotations denote the same elements as in Fig. 3)[1]

3.3.1 Language Design

Our solution automatically derived the technical meta-information from the FuMo DSL: the language contains first-class concepts for calls to other FuMos or parameter references; we listed such references in the appropriate section. Subversion managed the administrative meta-information.

We amended the prose text descriptions with explicit references to parameters (light gray rectangle in Fig. 6) and other FuMos (dark gray rectangle). Math formulas could be written in an embedded variant of the FuMo DSL (medium gray rectangle) [9]. Contrary to the Word version, we prefixed all references and formulas with specific characters akin to at-mentions (@) or hashtags (#) in social media. We applied the text formatting familiar from Word.

The original pseudo-code used some interesting depictions for typical control structures. We implemented them also for the FuMo DSL. They might inspire similar design for other languages. `switch`-like statements are shown as a table (see Fig. 10). The table header contained the condition. Each row represents one `case`. The condition fills the first column, the body is in the second column, and the third column contains a description or comment.

The end of a block repeated a rendering of the relevant details, e.g., the condition in an `if` (e.g., "Ende Falls *kz_rw_param = 12*" in Fig. 6).

We tried hard to shield users from technical details of the C implementation. Ideally, the generator should handle them. mbeddr takes care of some parts, e.g., generating an arrow or dot member access operator. For cases we could not handle automatically, we followed two approaches:

- Create a FuMo DSL concept, but do not allow the users to enter it. These concepts would only be used by the importer, e.g., explicit pointer de-referencing, explicit type casts, and explicit memory allocation.
- Use a generic *escape to C*: allows entering any mbeddr C code. Obviously, this code is completely in the responsibility of the user and outside any guarantees by the system. Examples include performance optimizations via pointer arithmetic and calls to undesired deprecated functions.

An internal accessory model provided stubs[4] for all library functions that need to be called by FuMo DSL code (see Sect. 2.3.1).

3.3.2 Language Implementation

We implemented prose text with embedded references and math by using *richtext* language.[5] We introduced specific characters (@ for references, # for math) as we had to switch into the appropriate context and show the correct auto-completion entries to the user. This turned out to be very helpful, as the users could directly type these symbols, and did not have to invoke any special action to switch context.

itemis has lots of experience with implementing procedural languages like Zurich's pseudo-code. Thus, the basic language implementation did not pose serious challenges. We spent some time inventorizing all pseudo-code concepts, as there was no formal definition. In discussions with the IT department, we clarified

[4]Referenceable placeholders for objects outside the model.

[5]Editor extension for arbitrary text intermixed with other elements [6].

ambiguous semantics. During *pattern recognition* (see Sect. 4.2.4), we identified additional domain-specific concepts and added them to the language.

Initially, we wanted to infer types in FuMo. After the first experimental import of the old code base, this turned out to be infeasible: in some cases, we could not assure the "direction" of type inference. As an example, a function parameter type should only be inferred from the function's body, not from its callers. Especially with the unsolved issue of function parameter type inference, any change to the code would trigger the typesystem to re-evaluate major parts of the code base. This overwhelmed MPS' typesystem implementation and would have rendered the editors unusable. However, the end-users were acquainted with explicit types anyway—they read them in C on a daily basis. Thus, explicit typing was acceptable; it was required in VADM in any case. Type checking could draw lots of inspiration from mbeddr and was implemented without notable hurdles.

4 Evaluation and Lessons Learned

MPS proved to be a good choice for this project. Projectional editors combined the familiar look to the end-users with semantically structured models. In combination with language extensibility, the editors enable new ways of dealing with imported legacy code. Generators assured consistent output.

We could mitigate MPS-related issues regarding editor and typesystem performance. We faced the biggest hurdles in the project with the import of the original code base. They were not specific to the MPS platform, but a mixture of generic legacy transformation and code-to-model challenges. The project's total effort amounted to roughly 40 person months.

4.1 Language Implementation

For both VADM and FuMo, projectional editors enabled a visual design very close to the original Word forms. Their implementation did not pose considerable challenges. MPS' language composition features enabled clean language design without too many compromises for backward compatibility, as we could defer edge cases to special constructs not accessible to end-users, or even to plain C models. Standard MPS features like technical references with forward and backward navigation, Subversion integration, and plain text intermixed with model elements contributed tremendously to the final result.

We experienced platform limitations in two areas: large editors tend to perform sluggishly, and we were not able to implement type inference as we planned. The table-heavy VADM editors posed the biggest editor performance issue. These editors barely interacted with the type-checking system, thus excluding it as potential performance issue. The original VADM was kept in few and large Word files.

Thankfully, the top-level VADM structures provided semantic borders to break the documents into several root nodes. The resulting editors performed reasonably. Regarding the typesystem issue, we abandoned the idea of heavy type inference. It might be possible to implement, but we decided not to spend the required effort. The impact on the result was acceptable, as our end-users were familiar with typed languages.

4.2 Import and Generation

Importing the original code was by far the most difficult part of the project. Even our original approach took way longer than anticipated (see Sect. 4.3), before we concluded it to be infeasible because of too long feedback loops. We eventually succeeded with the second approach (see Sect. 4.4), but again had to overcome unforeseen hurdles like running mainframe-targeted tests on a PC and semiautomatically analyzing test failures.

We suspected most of these issues to be typical of automated application modernization projects; the team had only limited experience in this field. Using domain-specific languages presumably did not add huge additional effort to the fundamental problem. On the contrary, the flexibility of language composition and interactive *pattern recognizers* (see Sect. 4.2.4) opened up new possible ways to deal with application modernization issues.

4.2.1 Import Source

We quickly concluded that we had to use the C code as base for the import: the C code was executed, so only this artifact was known to be correct. The IT department members knew a multitude of examples where the Word FuMo was outdated with respect to the implementation in C.

Another argument was technical: Word is hard to read programmatically, especially as we would need to preserve formatting (to identify parameter references) and indentation (for control structures). Zurich used advanced Word features like tables and track changes; this would require a very solid library to reliably access the document's contents.

The original Word VADM/FuMos contained also a prose text description. In some cases, this description was copied into the C source code as comments; in these cases, we could import the description. For others, we had to copy them by hand from Word. As a one-off action, this would take an acceptable effort of a couple of days.

4.2.2 Big Bang vs. Incremental Transformation

A *big bang* transformation processes the complete source at one point in time in its entirety. Before the transformation, only the source is used; afterwards, the complete source is discarded and the transformation outcome is the only usable artifact.

With an *incremental* transformation approach, parts of the source are transformed step by step. The source artifacts are valid for non-transformed parts, whereas the outcome is the only valid artifact for already processed parts.

We opted for a big bang approach to import for several reasons. The mbeddr C importer was responsible for importing the C source files into mbeddr C models. It had to resolve all references prior to import. The code base turned out to be highly coupled. There were no simple ways to cut the code base in independent sub-slices that would not reference each other. So we either had to import all at once or import overlapping sub-slices and merge them afterwards. We deemed merging to be much harder than an all-at-once import.

Both the IT department and the external service provider kept working on the source code during the project. This implies that in an incremental transformation approach, the C source files might have changed between the import of different sub-slices—rendering any kind of merge even harder. Moreover, it would have been very hard to synchronize changes in the C source files to already imported mbeddr C models, let alone FuMo DSL code. Applying some parts of a change in already imported models, and other parts in the C source files, does not seem feasible either.

The big bang approach implied we would never change imported models manually (besides development trials). All the improvements were applied to the original C source code. This provided some investment safeguard for Zurich: even if this project would have failed, they could still profit from the source code improvements.

4.2.3 Cleaning Up Sources

The C source code had been developed over several decades. Naturally, it accumulated technical debt like different implementation styles or workarounds that have never been fixed. One particular area to clean up was memory management: the source code used three different APIs to allocate and free memory. Zurich, itemis, and the external service provider unified them to one API. This revealed memory management issues like use-after-free or duplicate memory usage. Analyzing and resolving these issues took considerable effort.

The abovementioned cleanups are independent of targeting a DSL environment. More specifically to this target, we had to unify different ways to implement the same FuMo DSL construct in order to automatically recognize it. For example, the pseudo-code contained a `foreach`-loop concept. This can be implemented in C with a `for` loop and index variable, or a `while` loop and pointer arithmetic. If both patterns had been used a lot, we would recognize both. However, if it had been

implemented mostly with a `for` loop and the source had contained only a handful of `while`-loop variants, we would rewrite the latter.

4.2.4 Lifting from C to FuMo DSL

Lifting[8] describes the process of transforming rather low-level C code to semantically richer, more domain-specific language. Most of the concepts are very similar in C and FuMo DSL. A simple tree walker would process the C AST and create the corresponding FuMo DSL model. The tree walker would wrap any unrecognized element in an *escape to C* FuMo DSL concept.

Through manual inspection and interviews with IT department staff, we identified typical patterns in the C sources and how they mapped to FuMo DSL. We were very keen on matching domain-specific patterns, like formulas typical to insurance math, mortality table lookups, or access to Zurich-specific subsystems.

We implemented *pattern recognizers* to find these patterns during the C to FuMo DSL transformation. The tree walker incorporated the reliable (i.e., no false positives or negatives) recognizers. Less reliable recognizers were available as MPS intentions on the FuMo DSL. This combined the required manual assurance with easy application. It allowed continuous improvement both during our project and future development. Examples for pattern recognizers include `foreach` loops (see Fig. 7), VADM access, memory allocation, or pointer (de-)referencing.

The logical inverse of pattern recognizers are FuMo DSL to C generators. We developed them alongside the pattern recognizers.

4.2.5 Handling VADM Access

The actual VADM structures were expressed as C `structs`; importing them was straightforward. Importing their usage, however, was much more difficult. The runtime environment initialized one complete VADM structure (see Sect. 2.3.1). We spent serious effort to identify these access patterns and provide good abstractions in FuMo DSL. We did not want to expose the end-users to the intricacies of C pointer handling—they should be concerned about insurance business logic.

We found quite a few cases where we could not reliably recognize, lift, and generate the correct pointer access scheme; especially performance-optimized loop handling turned out to be problematic. There were even too many of them to be treated as edge case with an *escape to C*.

We resorted to explicit FuMo DSL constructs for pointer handling *for existing cases*: We created these concepts in the importer, but did not provide the end-user with a way to instantiate them. If future changes require performance optimizations, they need to be provided through support libraries.

```
public class ForEachStatementMapper extends AbstractFumoMapper {
  protected Pattern<BaseConcept> getPattern() {
    <ForStatement(condition: NotEqualsExpression(right: NullExpression()))>;
  }

  public node<> map(node<ForStatement> input) {
    node<Iterator> iterator = input.iterator;
    node<Expression> init = null;
    if (iterator.isInstanceOf(ForVarRef)) {
      node<ForVarRef> forVarRef = iterator:ForVarRef;
      init = forVarRef.init;
    } else if (iterator.isInstanceOf(ForVarDecl)) {
      node<ForVarDecl> forVarDecl = iterator:ForVarDecl;
      init = forVarDecl.init;
    }

    node<ForEachStatement> result = <ForEachStatement(
      body: # mapRecursive(input.body):StatementList,
      collection: # mapRecursive(init):Expression,
      variable: # mapRecursive(iterator):IVariableDefinitionOrReference
    )>;

    putInTrace(input, result);
    return result;
  }
}
```

Fig. 7 Screenshot of the pattern recognizer for `foreach` loops. *getPattern()* describes eligible partial C language models for this pattern recognizer; in this case, `for` loops with a condition of `...!= null`. *map()* converts the input pattern to the output. It handles both variable references (`if` clause) and variable declarations (`else` clause) in the loop's initializer. Finally, it constructs the partial FuMo output model and descents recursively into subtrees

4.3 First Importing Approach

Our original approach to import the existing C code base into FuMo models relied on Zurich's exhaustive test coverage. We expected fast turnarounds for each import–lift–generate–compile–test cycle. We anticipated issues in the transformation chain to be found by the C compiler and relied on test coverage to catch implementation differences.

The major steps were (reflected in Fig. 8):

1. Use mbeddr C importer to import the original C source code to equivalent mbeddr C models. This step should be a one-to-one transformation. Where needed, we would improve the importer iteratively.
2. Lift mbeddr C models to FuMo models, as described in Sect. 4.2.4. We expected to learn about new patterns and edge cases during development. They would be resolved by improving the FuMo lifter or replacing edge cases in the original source code (see Sect. 4.2.3).

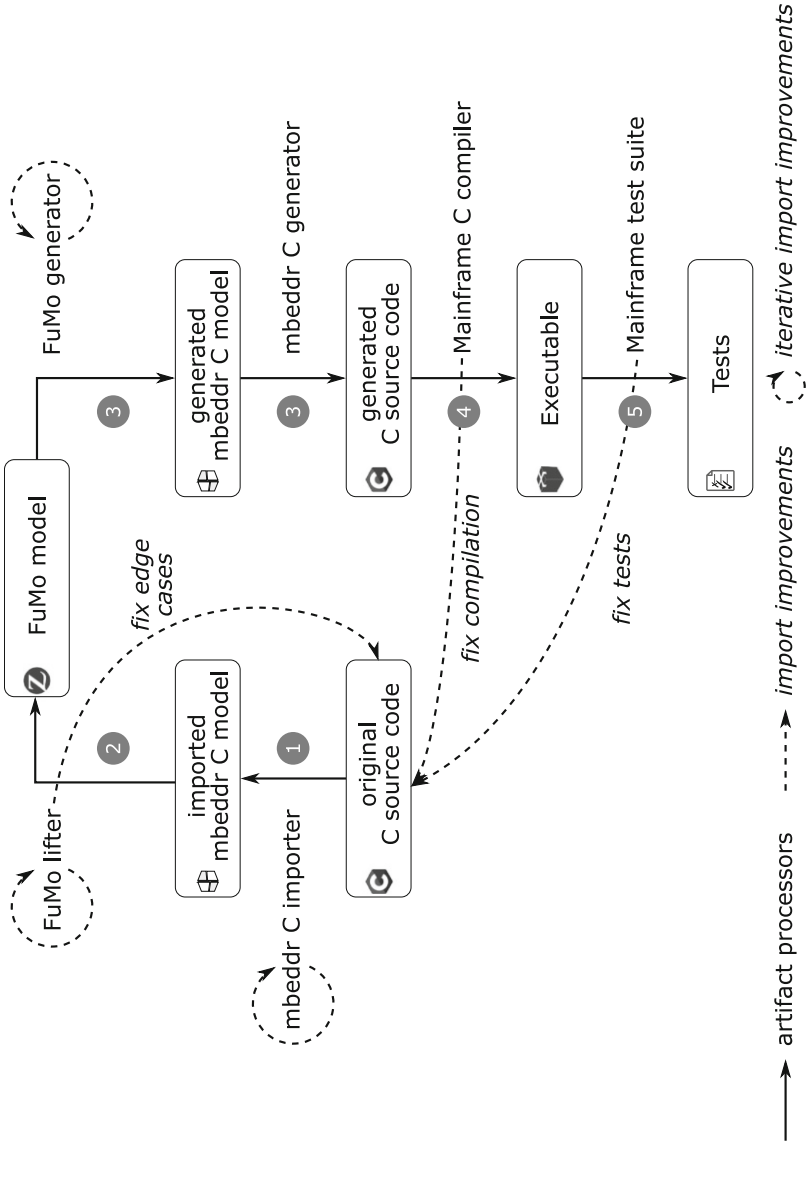


Fig. 8 Flow diagram of the first importing approach. Note the long feedback loop

3. Use the FuMo generator to transform FuMo models into mbeddr C models, and directly generate to C source code with the mbeddr C generator. The mbeddr generator has been used extensively in different contexts, so we expected little issues on that part.
4. With the help of the external service provider, compile the generated C source code on a mainframe. For any issues found by the compiler, we would adjust the original source code or the relevant processor along the transformation chain. We would iterate through these steps until the generated code can be compiled.
5. Execute the tests on a mainframe, as they relied on architecture specifics (see Sect. 2.4.2). We would apply the same fixes and iterations as for compiler issues until all tests pass.

This approach seems very naive in hindsight; we gained a lot of the experience presented here only after we had devised this approach. Based on our initial information, we assumed way faster processing times for almost every step. We anticipated a lot of similar errors at the beginning and assumed to converge quickly to edge cases—due to the highly automated processing chain. We ran into serious issues and ultimately failed with this approach.

The mbeddr C importer had been used in other projects before. However, C is very flexible. We encountered more parsing issues than anticipated, leading to more adjustment work. The importer took a lot longer to process the complete code base than we expected.

Originally, we expected to reuse the same PC-based C compiler as the IT department. This would have allowed us to compile the generated code ourselves, avoiding coordination overhead with the external service provider. Unfortunately, this compiler was unavailable outside Zurich, and setting up a usable development environment turned out to involve major effort. The PC-based C compilers were used to receive a lot more advancements than the mainframe C compiler. Thus, the compiler's error messages were not as helpful as we hoped.

The itemis development team lacked some advanced C knowledge. Even with some support by the external service provider, analyzing the compiler issues took some time.

During development, we did directly adjust mbeddr C models to analyze issues. But to assure our fixes would solve the problem, we had to adjust the original source code and rerun the complete import. At the beginning, a complete import took a weekend; later, we improved it to a night. In any case, the assurance took considerable time.

The test setup was not available outside Zurich's mainframe environment. Thus, a test cycle included shipping the generated code to Zurich, transferring it to the test system, running the tests, and getting the results back to the developers—not exactly a fast turnaround.

To summarize, this approach failed because it relied on fast turnarounds and checks in the model, or at latest by the compiler, to point out the majority of issues. We could not achieve the required turnaround speed and could recognize the issues only too late within an iteration.

4.4 Second Importing Approach

After having learned the lessons about turnaround speed and early issue recognition the hard way, we adjusted our approach. We could not reduce the turnaround time by splitting up the import in several sub-slices due to high coupling and the continued work by users, as described in Sect. 4.2.2.

Thus, we split up the single steps as far as possible and assured earliest possible feedback (steps are reflected in Fig. 9):

1. Make sure C import, generation, and test work (light gray background). After this step, we can trust the import processing chain from original C source code to mbeddr C model, and the generation processing chain from mbeddr C model via generated C source code, compiled with the mainframe compiler, to pass the mainframe test suite.
    a. Use mbeddr C importer to import the original C source code to equivalent mbeddr C models, and improve the mbeddr C importer where needed. This step stayed the same compared to the first importing approach.
    b. Directly generate the mbeddr C model to generated C source code, and compare the result with the original C source code. As the mbeddr C generator was quite mature, the only variable in this feedback loop was the mbeddr C importer. We adjusted the importer and, if required, the original C source code, until we encountered no meaningful differences between the original C source code and the generated one.
    c. Compile generated C source code on a mainframe. The generated C source code was *equivalent*, but not *identical* to the original one. The C language is very flexible, and especially compilers with a long history interpret details differently. With this feedback loop, we assured the mbeddr C generator and mainframe C compiler agreed on the same interpretation of C.
    d. Execute tests on a mainframe based on the generated C source code. Even if the mainframe C compiler had not flagged any issues with the generated C source code, it might still interpret the code differently than the original C source code. We eliminated that possibility by running the existing test suite on a mainframe based on the generated C source code.
2. Get faster, more expressive test results (medium gray background). This step aimed at decoupling the test execution from the external service provider, and provide more detailed insight why a test failed.
    a. Get tests running on a PC. We already managed to set up the PC-based C compilers during the first importing approach. However, we still depended on both the external service provider and a mainframe execution environment to run the test suite. With considerable effort, we found a way to run the test suite on a PC. Eventually, it produced the same test results as on a mainframe.
    b. Improve test results by trace logging. The test results did not provide any details on the failure reason, as explained in Sect. 2.4.2. Individually

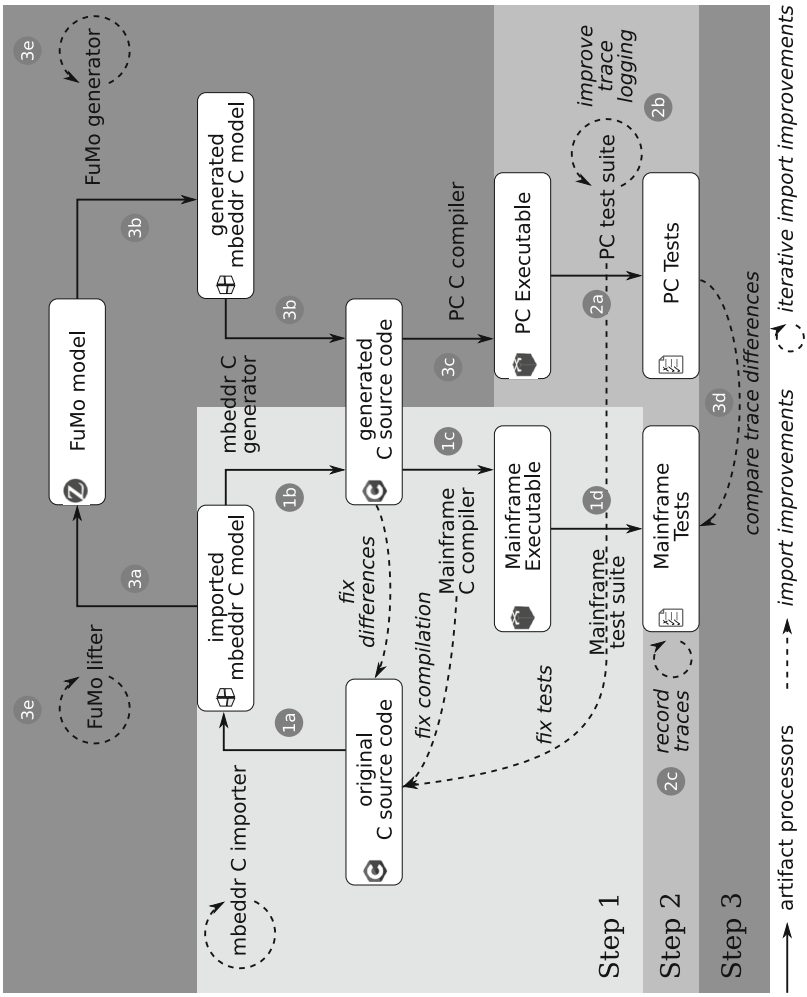


Fig. 9 Flow diagram of the second importing approach. Different background shades denote steps. Note separate, short feedback loops in each step

analyzing each test failure was completely unfeasible. Thus, we devised a semiautomated analysis: we wrote scripts that added trace logging statements to the C source code (refer to Sect. 4.5 for details).

c. Record traces from executing tests based on the original source code. With the trace logging scripts in place, we could apply them to the original C source code and record a *known good state* of trace logs. These logs formed the baseline to analyze deviations from running the test suite based on the generated C source code.

3. Get lifting and FuMo generation right (dark gray background). This step would add the missing parts of the processing chain compared to the first importing approach.

   a. Lift mbeddr C to FuMo DSL. Based on the validated mbeddr C model, we could focus on fine-tuning the FuMo lifter. Similar to the first importing approach, we extended and adjusted our pattern recognizers or fixed up edge cases in the original C source code.
   b. Generate to C. After the work on step 1b, we were sure the mbeddr C generator would produce the desired output. Thus, we only had to get the FuMo generator up to par.
   c. Run tests based on the generated code. Leveraging the effort of step 2a, we could run the test suite in a PC environment based on the generated C source code. We amended the generated C source code with the trace logging scripts.
   d. Compare traces with recorded traces to point out problematic parts. From step 2c, we had a baseline of trace logs. We compared this baseline with the trace logs from step 3c.
   e. Iteratively improve FuMo lifter and generator. At this point, the only source for trace log deviations could be the FuMo lifter or the FuMo generator. Thus, we adjusted both of them as necessary. We could execute all steps locally and iterate quickly. Finally, we met one of our original assumptions: We started with a lot of similar trace deviations and quickly converged on a rather small list of singular issues.

4.5 Analyzing Test Failures at Scale

We were fortunate enough to have a good test coverage to begin with, spanning thousands of tests. However, the test setup would only tell us *which* tests failed, with little hints on *why*. Due to the amount of tests and their execution time, it was infeasible to debug them one by one. Based on the available *known good* version of the source code, we devised a more efficient solution.

We wrote some scripts to automatically inject trace logging code at function calls, `if` branches, and other key points in the source code. We applied these scripts on both the original source code and our generated code. After running the tests with the original source code, we knew the desired execution path.

Simple text comparison of the desired execution path's trace logging with the one from our generated code revealed the places of concern. We could also cluster similar issues, as they would deviate from the desired execution path at the same point. This way, we reduced the number of required turnarounds by fixing the whole cluster at once and thus our total time to process all issues.

It turned out a lot of the failing tests were false failures. As described in Sect. 4.2.3, we streamlined different implementation styles. As a side effect, this changed how VADM attributes were used to store intermediate results. So after executing the FuMo under test, the resulting VADM structure contained different values than expected, and the test failed—but the differences only concerned insignificant attributes! Identical trace logs in a failing test proved to be a strong hint for those cases.

5 Conclusion

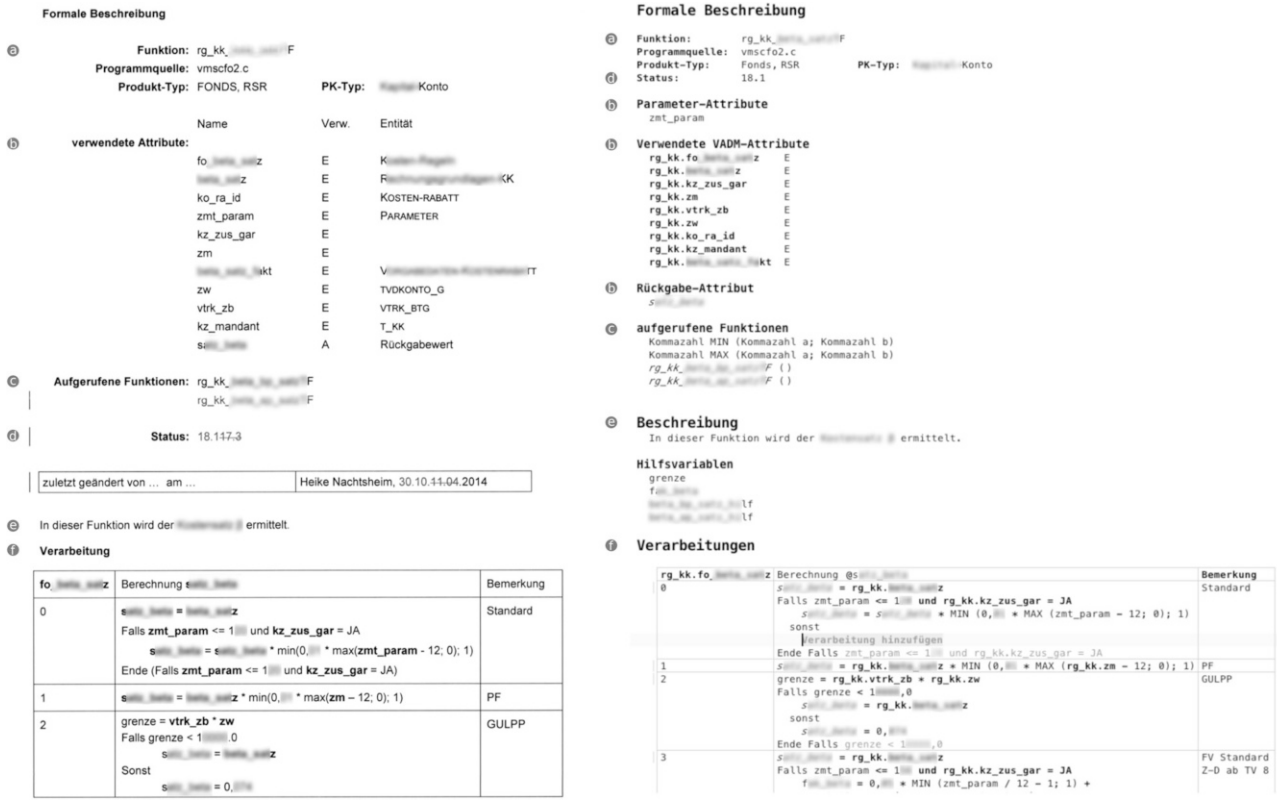
5.1 Technical Advantages and Shortcomings

MPS provided the means to go for a fully structured DSL while keeping the familiar document appearance (see Fig. 10). Language composition allowed well-designed and fine-grained decisions on first-class language concepts vs. backward-compatible compromises or low-level constructs. Intentions provided an easy way to integrate user-controlled pattern recognizers to higher-level language concepts.

We encountered performance issues if MPS showed big root nodes in the editor. Even more useful editors would have been possible if MPS provided cached access to *find all references* results. We did not succeed with implicit type inference, because we could not strictly separate scopes that should infer types from scopes that should only check type compliance.

Importing the existing code base posed the biggest challenge by far. We succeeded with a multistep approach that ensured early feedback on import issues. We strongly recommend short feedback loops for any large-scale model import, as processing time grows exponentially with every additional step, and the complexity even of simple transformations gets out of hand quickly.

Both non-incremental C source code import and insufficient model merge technologies forced us to only change the original C sources and reimport the complete code every time. Improvements in both fields would considerably simplify similar challenges.



Formale Beschreibung

ⓐ **Funktion:** rg_kk_[blurred]F
Programmquelle: vmscfo2.c
Produkt-Typ: FONDS, RSR **PK-Typ:** [blurred]Konto

	Name	Verw.	Entität
ⓑ **verwendete Attribute:**			
	fo_[blurred]z	E	K[blurred]
	[blurred]z	E	R[blurred]KK
	ko_ra_id	E	KOSTEN-RABATT
	zmt_param	E	PARAMETER
	kz_zus_gar	E	
	zm	E	
	[blurred]kt	E	V[blurred]T
	zw	E	TVDKONTO_G
	vtrk_zb	E	VTRK_BTG
	kz_mandant	E	T_KK
	s[blurred]	A	Rückgabewert

ⓒ **Aufgerufene Funktionen:** rg_kk_[blurred]F
rg_kk_[blurred]F

ⓓ **Status:** 18.~~117~~3

zuletzt geändert von … am …	Heike Nachtsheim, 30.10.~~11~~.04.2014

ⓔ In dieser Funktion wird der [blurred] ermittelt.

ⓕ **Verarbeitung**

fo_[blurred]z	Berechnung s[blurred]	Bemerkung
0	s[blurred] = [blurred]z Falls **zmt_param** <= 1[blurred] und **kz_zus_gar** = JA s[blurred] = s[blurred] * min(0,[blurred] * max(**zmt_param** - 12; 0); 1) Ende (Falls **zmt_param** <= 1[blurred] und **kz_zus_gar** = JA)	Standard
1	s[blurred] = [blurred]z * min(0,[blurred] * max(**zm** – 12; 0); 1)	PF
2	grenze = **vtrk_zb** * **zw** Falls grenze < 1[blurred].0 s[blurred] = [blurred]z Sonst s[blurred] = 0,[blurred]	GULPP

Formale Beschreibung

ⓐ **Funktion:** rg_kk_[blurred]F
Programmquelle: vmscfo2.c
Produkt-Typ: Fonds, RSR **PK-Typ:** [blurred]Konto
ⓖ **Status:** 18.1

ⓑ **Parameter-Attribute**
zmt_param

ⓑ **Verwendete VADM-Attribute**
rg_kk.fo[blurred]z E
rg_kk.[blurred]z E
rg_kk.kz_zus_gar E
rg_kk.zm E
rg_kk.vtrk_zb E
rg_kk.zw E
rg_kk.ko_ra_id E
rg_kk.kz_mandant E
rg_kk.[blurred]kt E

ⓑ **Rückgabe-Attribut**
s[blurred]

ⓒ **aufgerufene Funktionen**
Kommazahl MIN (Kommazahl a; Kommazahl b)
Kommazahl MAX (Kommazahl a; Kommazahl b)
rg_kk_[blurred]F ()
rg_kk_[blurred]F ()

ⓔ **Beschreibung**
In dieser Funktion wird der [blurred] ermittelt.

Hilfsvariablen
grenze
f[blurred]
[blurred]lf
[blurred]lf

ⓕ **Verarbeitungen**

rg_kk.fo[blurred]z	Berechnung @s[blurred]	**Bemerkung**
0	s[blurred] = **rg_kk.**[blurred]z Falls zmt_param <= 1[blurred] **und rg_kk.kz_zus_gar = JA** s[blurred] = s[blurred] * MIN (0,[blurred] * MAX (zmt_param – 12; 0); 1) sonst Verarbeitung hinzufügen Ende Falls zmt_param <= 1[blurred] und rg_kk.kz_zus_gar = JA	Standard
1	s[blurred] = **rg_kk.**[blurred]z * MIN (0,[blurred] * MAX (**rg_kk.zm** – 12; 0); 1)	PF
2	grenze = **rg_kk.vtrk_zb** * **rg_kk.zw** Falls grenze < 1[blurred],0 s[blurred] = **rg_kk.**[blurred]z sonst s[blurred] = 0,[blurred] Ende Falls grenze < 1[blurred],0	GULPP
3	s[blurred] = **rg_kk.**[blurred]z Falls zmt_param <= 1[blurred] **und rg_kk.kz_zus_gar = JA** f[blurred] = 0,[blurred] * MIN (zmt_param / 12 – 1; 1) +	FV Standard Z-D ab TV 8

Fig. 10 Comparison of Word (left) and MPS FuMo (right), including *track changes* marks in Word, and a `switch`-like statement. The annotations point out equivalent contents on both sides. (Partially blurred for confidentiality)[1]

5.2 Project Results

The users in the IT department are very satisfied with the MPS-based solution. We consider their early involvement, continuous feedback during development, and early training for both MPS and Subversion vital for the success of the project.

The users profit from guaranteed consistency of the FuMo DSL code they are working with and the executed code. The turnaround time between a change in a FuMo and testable code is reduced by several orders of magnitude—from days or weeks to seconds.

Complete version control of all artifacts through Subversion assures safe processing and storage of changes. It streamlines collaboration, both within the IT department and with the external service provider.

Shorter turnaround time and reduced communication overhead lead to higher efficiency and fewer misunderstandings, ultimately speeding up time-to-market and lowering the defect rate.

Both the IT department and the external service provider can focus on their field of expertise: insurance domain knowledge and operational/nonfunctional aspects, respectively.

The MPS-based FuMos are in production for more than 2 years. Zurich and itemis continue to improve the system, migrate it to new MPS versions, and adjust it to new requirements.

References

1. CollabNet Inc., et. al.: Apache Subversion (2021). https://subversion.apache.org/
2. Grosche, A., Igel, B., Spinczyk, O.: Exploiting modular language extensions in legacy c code: An automotive case study. In: I. Schaefer, D. Karagiannis, A. Vogelsang, D. Méndez, C. Seidl (eds.) Modellierung 2018, pp. 103–118. Gesellschaft für Informatik e.V., Bonn (2018)
3. itemis AG: MPS Extensions (2021). https://www.itemis.com/
4. itemis AG, et. al.: mbeddr (2021). http://mbeddr.com/
5. itemis AG, et. al.: mbeddr Platform (2021). http://mbeddr.com/platform.html
6. JetBrains s.r.o., et. al.: MPS Extensions (2021). https://jetbrains.github.io/MPS-extensions/
7. Pilato, C.M., Collins-Sussman, B., Fitzpatrick, B.W.: Version control with subversion: next generation open source version control. O'Reilly Media, Inc. (2008)
8. Tomassetti, F., Ratiu, D.: Extracting variability from c and lifting it to mbeddr. In: Proceedings of the International Workshop on Reverse Variability Engineering (2013)
9. Voelter, M.: Integrating prose as first-class citizens with models and code. In: MPM@ MoDELS, pp. 17–26. Citeseer (2013)
10. Voelter, M., Ratiu, D., Schaetz, B., Kolb, B.: mbeddr: an extensible c-based programming language and ide for embedded systems. In: Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, pp. 121–140 (2012)
11. Zürich Beteiligungs-Aktiengesellschaft (Deutschland): Zurich Versicherung (2021). https://www.zurich.de/