Experiences from Reengineering and Modularizing a Legacy Software Generator with a Projectional Language Workbench

Max Lillack
Leipzig University, Germany

Thorsten Berger
Chalmers | University of Gothenburg, Sweden

Regina Hebig
Chalmers | University of Gothenburg, Sweden

ABSTRACT

We present a case study of migrating a legacy language infrastructure and its codebase to a projectional language workbench. Our subject is the generator tool ADS used for generating COBOL code for critical software systems. We decompose the ADS language into smaller sub-languages, which we implement as individual DSLs in the projectional language workbench JetBrains Meta Programming System (MPS). Our focus is on ADS' preprocessor sub-language, used to realize static variability by conditionally including or parameterizing target code. The modularization of ADS supports future extensions and tailoring the language infrastructure to the needs of individual customers. We re-implement the generation process of target code as chained model-to-model and model-to-text transformations. For migrating existing ADS code, we implement an importer relying on a parser in order to create a model in MPS. We validate the approach using an ADS codebase for handling car registrations in the Netherlands. Our case study shows the feasibility and benefits (e.g., language extensibility and modern editors) of the migration, but also smaller caveats (e.g., small syntax adaptations, the necessity of import tools, and providing training to developers). Our experiences are useful for practitioners attempting a similar migration of legacy generators to a projectional language workbench.

CCS Concepts

•**Software and its engineering** → *Extensible languages; Source code generation; Maintaining software;*

1. INTRODUCTION

Preprocessors are common and established tools to extend existing programming languages. They can be found for many programming languages, either as part of the compiler infrastructure (e.g., the C preprocessor) or as a standalone tool (e.g., Antenna, M4). Among others, they are a popular mechanism to implement compile-time variability. Together with other features that enhance the capabilities of their underlying target languages with commonly needed functionality—such as modularization, control-flow or I/O statements—they provide the basis for comprehensive software generators.

In some domains, such generators have become almost as important as the target language. Consider the banking and insurance domain, which relies largely on COBOL or PL/I[1], but has developed comprehensive generators for coping with the various shortcomings of COBOL. One such generator is the preprocessor-based Application Development System (ADS)—the subject of our case study.

While there is a strong need to maintain systems developed with these legacy generators, there is also a need for modernization of these systems and, as a consequence, for modernizing the generator itself. Traditional modernization scenarios, such as those described by the OMG [1], focus on the platform, architecture or programming language of a single system. The existence of legacy generators leads to additional modernization goals: (i) improving the modularization and extensibility of software generators, and (ii) handling variability—expressed in the preprocessor language—more explicitly, in order to enable a combination with more modern variability implementation and analysis techniques. In fact, improving modularization also facilitates a later modernization of the underlying target language (e.g., COBOL). Yet, migrating such legacy language infrastructures is a complex task for companies, requiring incremental approaches starting with modernizing the generator. Unfortunately, only few empirical data and experience reports are available about the migration of such language infrastructures [2, 7].

We present a case study of modernizing the legacy generator tool ADS. It has roots in the 1970s, but is still used today by many companies in the banking and insurance domain, to construct large (more than a million lines of code) and critical systems. We migrate a part of a large generator system for creating a COBOL application that is used to manage Dutch car registrations. ADS consists of a complex *preprocessor* language and more than 20 *sub-generators*. The former is important for variability—to generate variants for different users or target platforms by conditionally including or parameterizing code. The latter provide commonly needed functionality lacking in COBOL, such as program structuring, I/O functionality, and database access. Other sub-generators help to design screens and reports or to create documentation. The preprocessor and the various sub-generators can be seen as different languages, but all are mingled together in ADS.

We conduct the case study in close cooperation with the

[1]Another legacy language used in the banking domain.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC '16, September 16 - 23, 2016, Beijing, China

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4050-2/16/09...$15.00

DOI: http://dx.doi.org/10.1145/2934466.2962733

provider of ADS Delta Software Technology[2] and two large industrial ADS users. We gather information about the existing ADS systems and discuss requirements with the tool provider. The latter has additional expectations to the modernization: obtaining slight improvements of ADS' syntax; having the ability to later complement (or completely substitute) the generation of COBOL with other languages, such as Java or C#; and to get better tooling—including debugging and better editor support (e.g., code completion).

We show how the ADS infrastructure can be migrated to a modern language workbench [6, 8, 15]. Language workbenches are tools for engineering and using (domain-specific) languages. We chose JetBrains Meta Programming System (MPS) [17], which relies on a projectional editor [3, 19] offering powerful facilities for language composition and flexible notations, and is therefore well-suited for realizing a modularization of ADS into sub-languages and later extending and replacing them. Our strategy is to decompose ADS into smaller DSLs and to re-implement ADS' program-generation logic as modular model transformations.

We evaluate the feasibility and potential benefits of using MPS, by realizing a tool to transform existing ADS code into MPS, and by validating the transformation. In summary, we contribute:

- Challenges and requirements for migrating ADS.
- A migration approach to MPS, including: (i) an implementation of a subset of the ADS languages, (ii) a transformation of ADS source files into MPS' representation, and (iii) a validation of the approach.
- A set of lessons learned.

By reporting our experiences and compiling a list of lessons learned, we aim at supporting similar migration approaches and help researchers and practitioners develop better migration techniques. As we will show, MPS can be used to re-create ADS in an efficient way, and the result is a suitable base for future maintenance as well as further modernization steps, such as replacing COBOL as the target language.

2. BACKGROUND

We now briefly introduce the software generator ADS and our target language workbench JetBrains MPS.

2.1 ADS

Fig. 1 shows a small example of ADS. To the left, we see a source file. Lines 1 and 4, as well as the variable reference `#01` in Line 6, are part of the preprocessor language, which is executed at generation time. Lines 2–3 contain embedded target language code (COBOL), which is directly generated into the output file. Lines 5–9 show a sub-generator for program logic. From the input of this sub-generator, source code is generated into the output file.

The code in the middle and on the right-hand side of Fig. 1 illustrates the two phases of the generator—preprocessing and applying the sub-generators. Thus, to replace this legacy generator infrastructure, it is necessary to create a tool that supports ADS' two phases of generation.

Before the migration of the generator, we already built tools to *analyze* existing ADS code, which helped us understanding the use of ADS. Based on these previous analyses, we can decide which sub-generators of the legacy generator are still in use and therefore need to be supported by the new generator. Specifically, we identify code where new languages features, such as logical operators, could be used (cf. Sec. 5).

2.2 Language Workbenches

Language workbenches provide two main functionalities, both needed for our migration. First, they support the *design* of (mostly textual) DSLs, including the definition of concrete and abstract syntax, together with the static and dynamic (execution) semantics [15]. Second, they support the *use* of DSLs for software development [5], including navigating, editing, debugging, and executing DSLs.

From the perspective of the generator tool vendor, the projectional language workbench makes it easy to create and integrate new sub-generators (new DSLs). For the users who use the generator tool to generate different variants of COBOL or PL/I source code, the language workbench provides full IDE support to develop programs.

The advantage of using a language workbench is that we can rely on existing and proven tools. Additional functionality can be realized using extension mechanisms, such as MPS' function hooks and plug-ins. Any improvement to the language workbench can be used immediately for the generator, and any documentation and teaching material can be reused.

2.3 JetBrains MPS

MPS is a language workbench that relies on a projectional editor—both for designing and for using DSLs. Such an editor allows users to directly work on the abstract syntax tree of a domain-specific model or program. No parsing is involved. While users still see the concrete syntax, their editing activities directly change the tree by adding, moving, or removing nodes among other. This allows unlimited language composition (no grammar disambiguation is needed) and the use of different syntaxes, such as textual and graphical ones, in the same file. For one abstract syntax of a language, multiple so-called projection rules can be defined for showing the language's concrete syntax in the editor.

The usability of projectional editors, providing a different editing experience, has long been questioned, but recent improvements in MPS show that users can efficiently write code in it [3, 19].

To define languages, MPS provides a multitude of options. These are grouped as *aspects*, whereas some are optional (e.g., refactorings) and some are mandatory (e.g., the definition of statement types). For us, the following four *aspects* are essential: The *Structure* aspect describes a language's possible elements (called *concepts*) and their relation. This aspect is commonly known as the metamodel or abstract syntax, as it describes the tree structure of a valid program, but not the syntax of the elements. The *Editor* describes the concrete syntax(es)—views rendered by the editor—of the language. That is, this aspect is used to define projection rules for the language concepts. Finally, the semantics of a language is defined in the *Generator* aspect using model-to-model transformations and the *textGen* aspect using model-to-text transformations. MPS supports nine more aspects for defining languages, but these were not necessary for our migration.

3. CHALLENGES AND REQUIREMENTS

We identify the following four challenges based on our own experiences and those of our three partner companies. We

[2]http://delta-software.com

Original File

```
01 .IF-01.EQ.X
02    MOVE 1 TO XYZ-1
03    MOVE 1 TO XYZ-2
04 .IFEND
05 .SPP
06   DO LOOP-TRT VARY I1 FROM 1 TO #01.
07      MOVE 1 TO XYZ-I1
08   END LOOP-TRT
09 .END
```

Preprocessed File

```
   MOVE 1 TO XYZ-1
   MOVE 1 TO XYZ-2

.SPP
  DO LOOP-TRT VARY I1 FROM 1 TO X.
     MOVE 1 TO XYZ-I1
  END LOOP-TRT
.END
```

Result

```
   MOVE 1 TO XYZ-1
   MOVE 1 TO XYZ-2

   MOVE X TO DX-3.
   MOVE 1 TO I1.
   GO TO DX-5-1T.
DX-5-1.
   ADD 1 TO I1.
DX-5-1T.
   IF I1 GREATER DX-3 GO TO DX-7.
     MOVE 1 TO XYZ-I1
   GO TO DX-5-1.
DX-7.
```

Preprocessor language
Embedded target language
Sub-Generator language

Figure 1: A small ADS example: original file (left), file after preprocessing (middle), and file after running the sub-generators (right)

formulate requirements based on the challenges and will get back to them in the discussion of our solution in Sec. 5.

3.1 Incremental Modernization

Legacy systems are a known problem for development and maintenance, but are also a reliable foundation for business's functionality and many surrounding systems [10]. Legacy systems typically consist of multiple technologies and tools. Besides the generator tool, there are middleware systems, compilers, user interfaces, and interfaces to other systems. Each of these components could benefit from a modernization, but replacing all parts at once was not a viable option for our partners:

R1. Conduct an incremental modernization.

In fact, a focused modernization project is more consistent with typical requirements for software projects (time, budget, acceptance) even if this limits the potentials benefits. For instance, users keep their existing COBOL code base, but make changes to the environment: they may run this code on .NET rather than mainframe or may exchange the database.

3.2 Legacy Assets

Our partner companies are very sensitive to changes in core applications. Such changes can be risky and costly and need to be balanced with the potential benefits. Users of ADS spent years developing and testing their applications. To keep this investment, we need to provide an automated way to tranform existing code-bases:

R2. Reuse legacy assets. Re-writing existing code in the new technology is too risky and expensive.

For any such transformation of legacy assets we need to:

R3. Ensure the correctness of the transformation.

Although a modernization based on the *generated* source code seems possible, it is much more desirable to keep the variability and the design encoded in the generators:

R4. Maintain existing variability. Users need to be able to generate the same variants that could be generated before.

Finally, we need to address shortcomings of the existing tools or languages. The legacy editor only supports syntax highlighting for the preprocessor and some keywords of the sub-generator languages:

R5. Provide modern editor support for the sub-generators.

The ADS language has some technical limitations—for instance, missing logical operators or strict naming schemes, which sometimes prevent meaningful variable names:

R6. Allow new language features to remove existing language limitations.

3.3 Acceptance

Besides technical challenges, organizational factors influence the success of a modernization project. One such factor is the availability of developers for the legacy technology. Since the companies want to keep such developers with experience in the legacy system, we need to create a solution that will be accepted both by developers with a background in legacy technologies and developers experienced with modern technology. Documentation should help to understand the structure of the new system:

R7. Automatically generate documentation about the source files and their key properties (e.g., their interfaces).

3.4 Flexibility

The generator tool supports many sub-generators, but customers typically use only a subset:

R8. Modularize the ADS language to provide tailored subsets of its sub-generators to customers.

Finally, it is difficult to foresee the next steps in the overall modernization process—for instance, whether COBOL will be replaced with Java or C#. The modernized generator should be a useful intermediate step and still be flexible enough to enable more modernization steps later:

R9. Choose a language implementation that can be used to generate arbitrary target languages.

4. APPROACH

Our migration approach comprises the following steps as summarized in Fig. 2.

First, we re-engineer the legacy generator tool, including the preprocessor DSL and the various sub-generator DSLs, in MPS. We design the languages (step *Language Design*) including their abstract and concrete syntax, and then implement the actual functionality of the generator: pre-processing, executing the sub-generators, and generating HTML documentation files (step *Transformations*).

Second, we migrate existing code to the target platform MPS. We implement an import tool relying on parsing exist-

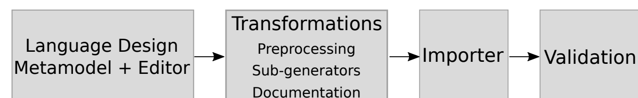

Figure 2: Steps of the approach to setup the new generator and migrate existing programs

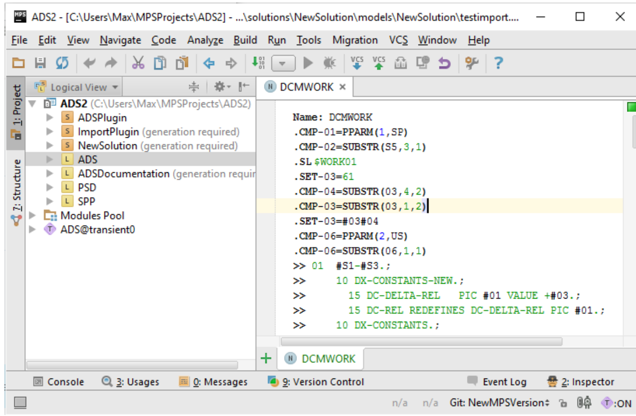

Figure 3: Screenshot of the resulting editor

ing code (step *Importer*) and creating instances of the new DSLs using MPS' API. Recall that projectional editing does not rely on parsing: user's editing action directly change a rich AST, which is not serialized/deserialized using the concrete syntax anymore, but in an XML format, so dedicated importers need to be written. We then validate the tranformation of existing code (step *Validation*).

4.1 Realization of the Generator

Fig. 3 shows a screenshot of the new generator tool. To the left, it shows the different projects to define the languages, the import plugin and a project with the imported files. To the right, we see the projectional editor showing the use of the preprocessor language. In the following, we present our implementation in detail.

4.1.1 Language Design

We define ADS' preprocessor language and legacy sub-generators as individual languages (DSLs) in MPS. The preprocessor language is our main focus, since it is one of the most important parts of ADS and used for handling variability, among other uses. It is also more complex than the C preprocessor (22 types of statements instead of 10, explained shortly) and rather similar to a procedural programming language. The sub-generator languages cover domain-specific functionality. We implemented features from SPP, a sub-generator to perform *structured programming*, providing higher-level control structures missing in COBOL (e.g., loops and switch statements) that are transformed into lower-level COBOL statements. The other sub-generator we implemented is PSD, with which a developer can generate a data-driven program skeleton based on a data specification.

We do not implement the target language (COBOL) as a DSL. It is instead represented as lines of text (using a simple concept holding the text value)—so COBOL is plain text and no improved editor support is available. In Sec. 5 we discuss efforts and benefits of realizing COBOL in MPS, which is subject to future work.

We use multiple sources to learn about the languages. First, the user manual describes names and syntax of all language features. Second, we run an existing parser for the preprocessor language on small code examples and inspect its parse tree. After getting an initial set of statements, we iteratively check with a larger sample that we use for the actual migration (see Sec. 4.2.2). We obtain 22 different statement types for the preprocessor language, which also includes typical library functions (e.g., substring). In contrast, CPP with about ten statement types is much simpler. These parse tree elements constitute language concepts to be defined in MPS. Finally, for the sub-generator SPP we obtain seven statements, and only one for the sub-generator PSD—implementing this subset of the sub-generators sufficed for transforming our existing ADS code (see Sec. 4.2).

Abstract Syntax. We create the metamodels (MPS' *Structure* aspect) manually, but mostly as a mechanical transformation based on the insights from the parser and the user manual, keeping concept names and the language structure. For instance, the preprocessor language has a root, which contains a set of statements, and statements may be assignments, loops, and so on.

For the generation process, where we needed to save the preprocessed model in an intermediate model, we extended the preprocessor metamodel with five more concepts (e.g., the location concept, explained in Sec. 4.1.2).

Concrete Syntax. We design the concrete syntax (MPS' *Editor* aspect) similar to the legacy syntax to gain a level of familiarity for existing developers. Using the existing syntax as a start makes it easy to visually assess the migration, because we can easily view legacy and new source next to each other. Later, the syntax can be changed without any influence on other aspects of the language (e.g., metamodel).

Fig. 4 shows an excerpt of a generator file using the original source code (left) and as seen in MPS after the migration (right). The abstract syntax is clearly the same, so it is easy to see how statements are mapped to the new system. If we compare the snippets in more detail, we see some subtle changes in the syntax, however. Some changes (e.g., the `>>`) will be explained in Sec. 6, others are just a matter of taste.

MPS supports the creation of multiple views for the same model. This way, we could define different concrete syntaxes for developers. For example, the legacy syntax of the comparison operator uses `EQ`, while some developers might prefer `==`. Developers could select the view they prefer.

4.1.2 Transformations

The generation process defines the semantics of the languages. Using MPS' aspects *Generator* and *TextGen* we define model-to-model and model-to-text transformations. The model-to-model transformations execute the preprocessor and create an intermediate model. The model-to-text transformations generate the output from the sub-generators and create the final output file.

Preprocessor. We implement the preprocessor as a custom in-place model-to-model transformation of the ADS model.

```
.SL=$WORK01                                  .SL $WORK01
 01   DX-TAP.                                >> 01   DX-TAP.;
.CMP-03=LENGTH(S1),                          .CMP-03=LENGTH(S1)
.IF-03.GT.8                                  .IF ( #03 . GT . 8 )
.CMP-03=SUBSTR(S1,1,8)                         .CMP-03=SUBSTR(S1,1,8)
.IFELSE                                      .ELSE
.SET-03=#S1,                                   .SET-03=#S1
.IFEND                                       .ENDIF
     15 DX-TAP-PROGID    '#03'.              >>      15 DX-TAP-PROGID   '#03'.;
     15 DX-TAP-GENDAT    '#UA'.              >>      15 DX-TAP-GENDAT   '#UA'.;
```

Figure 4: Example of legacy (left) and migrated (right) syntax

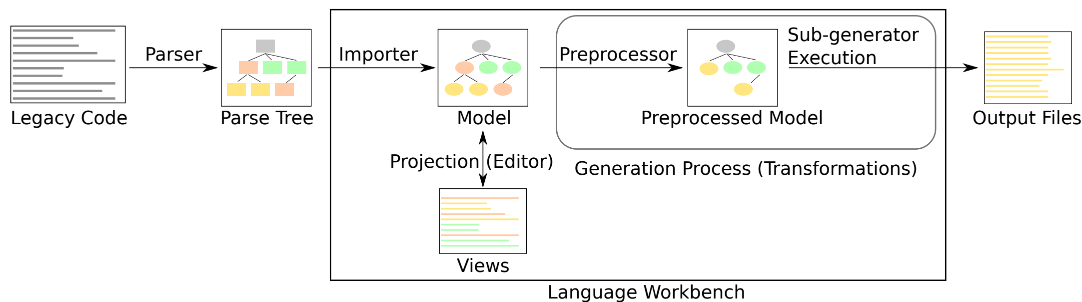

Figure 5: Overview of the migration and execution of the new generator

For instance, all preprocessor variable references are resolved to their actual values, and conditionals are resolved (e.g., the AST sub-tree "if x then COBOL-A else COBOL-B" is replaced with a "COBOL-A" node if x is true). This transformation belongs to MPS' *Generator* aspect and is called when a generation process is started.

The implementation (slightly more than 1,000 lines of Java code) relies on the standard Java library as much as possible—for instance, ADS' substring function is simply implemented by calling Java's `String.substring()`.

The preprocessing does not change the structure of the model, only replaces nodes in the AST. This modified model is then passed to a further model-to-model transformation.

Intermediate Model. We create an intermediate model with a different order of the model elements. Often, the target language prescribes a certain layout of the source code, which is often not consistent with the design in the generator. For instance, a COBOL file contains a dedicated header area in which variable declarations need to be put. To declare and use variables at the same place, the preprocessor language provides the concept of *locations*. A location can be declared—denoting where code belonging to it should be generated—and referenced throughout the ADS file. The latter wraps code, which is then moved (generated) into the declared target location.

Sub-Generators. For the sub-generator languages, we implement model-to-text transformation to generate COBOL source code from their abstract input (the intermediate model). In the example from Fig. 1 a `DO` loop from the SPP sub-generator is transformed into COBOL code using only `IF` and `GOTO`. Finally, the model-to-text transformation will create the final output file from the intermediate model and the results from the sub-generators.

Documentation. Generating documentation similar to Javadoc—listing all available migrated ADS modules and their API in an HTML file—was simple, since the required information can easily be extracted from the MPS model. So we added another model-to-model transformation for the same modules. MPS already provides a DSL for XML, which we use as the target metamodel for this transformation in order to obtain an XHTML file.

4.2 Migration of Existing ADS Code

In this subsection, we present an importer we wrote for migrating existing ADS code, and show the actual migration including a validation of it. Fig. 5 shows an overview of the final result, starting with the migration, then the new generator and the validation.

4.2.1 Import

Importing legacy assets into the language workbench is not a prominent feature of MPS. MPS usually persists models in its own XML format, which should not be changed by external tools. Similar migration projects needed to create custom importers [18].

There are two ways to design an import of legacy source code to MPS. First, *custom persistence* allows overriding the default persistence logic for our own format, such as ADS' legacy syntax. Second, we could programmatically create the models within MPS using its API.

We follow the second approach, since we only need to import the existing code once. Furthermore, any future language changes do not require adapting the persistence logic. For our importer we use an existing parser from our industry partner, which creates an XML-based parse tree. We traverse this parse tree and create a corresponding MPS node for each visited entity.

Recall that the generator uses a two-stage generation process. The preprocessor language in the first stage can be used to manipulate the input of the sub-generators in the second stage. This creates the problem of parsing unpreprocessed code [9]. Technically, the preprocessor language could be used to arbitrarily construct inputs for the second stage. Especially undisciplined annotations [12], where the preprocessor language is used to optionally include fine-grained source code that cross-cutts statements of the sub-generator inputs, are a problem for parsing. In practice, however, ADS' preprocessor statements are mostly used to parameterize identifiers or to optionally include complete statements. This disciplined usage allows us to rely on a simple parser for the unpreprocessed code.

4.2.2 Validation

We apply the migration to two kinds of systems: (i) library files that are implemented and delivered from the company providing the generator and (ii) systems that are implemented by users/customers.

We validate our approach to ensure the correctness of the new tool. To test the preprocessor language, we use the library files because they heavily use the preprocessor. We import all the existing files, generate them using the default configuration, and compare the results with the version generated by the original tool for the same input.

Library Files. ADS internally uses a set of macro files written almost exclusively using the preprocessor language. These are part of the implementation of a sub-generator, but can also be used by developers as a library for commonly

Table 1: **Use of statement types in library and user files**

Type of Statements	Library	User
Preprocessor	11,971 (56 %)	834 (16 %)
FGEN sub-generator	169 (1 %)	1,837 (34 %)
SPP sub-generator	489 (2 %)	1,455 (27 %)
PSD sub-generator	0 (0 %)	86 (2 %)
FILE sub-generator	17 (0 %)	70 (1 %)
Target language	8,846 (41 %)	1,071 (20 %)
Number of Files	222	68
Lines of Code	24,413	5,866

needed functionality. In total, there are 228 library files[3] with more than 24,000 lines of code (excluding comments). These library files are a good benchmark, as they use almost every feature of the preprocessor language.

User Files. User files include all the inputs for the generators created by the developers using ADS. We obtained files from one of our partner companies developing a large car-registration system used by the Dutch authorities. The complete system consists of thousands of programs, each of which are generated with ADS and use multiple files during generation. Of all these programs, our partner selected a representative, but relatively small set of programs and corresponding macro files we could use.

Summary of Migrated Files. Table 1 summarizes the language use in our two validation samples. It shows the number of statements written using the preprocessor language and the number of lines written in each of the sub-generator languages. The files only use the sub-generators FGEN, SPP, PSD and FILE. The row *Target language* shows the number of lines that are written into the output file: these lines may be parameterized with variables from the preprocessor language. The total file count is not representative, the actual number of files in the user system is much larger but our study contains only a sample of them.

The library files consist almost only of the preprocessor language (56 %) and embedded target language (41 %). There are only few instances where sub-generators are used. On the other side, user files make heavy use of the sub-generators, only (20 %) of the code is embedded target language code, and everything else is code written in a sub-generator language or the preprocessor language. Even though the sub-generator PSD is only used in a few lines, it is still important because it is used in almost every generator run once. Currently, we only import the parts of the sub-generators PSD and SPP we implemented in MPS.

5. DISCUSSION

We now briefly discuss our solution with respect to the key requirements collected from the different stakeholders.

R1 (Incremental Modernization). Our solution is designed as a drop-in replacement for the legacy generator tool, so it requires no changes in any connected system. A necessary one-time migration is fully automatic and requires no effort.

[3]We could not analyze six of the library files due to parsing errors.

R2 (Legacy Assets). With our implementation of an importer we are able to transform existing source code to the new tool and keep its functionality. We implemented all features of the preprocessor language and parts of two often used sub-generator languages: SPP and PSD.

R4 (Variability). As a consequence from R2, we keep the variability implemented with the preprocessor language. The modular design of the language workbench makes it possible to introduce a more explicit handling of variability, such as using a variability DSL as demonstrated in mbeddr[18]. Such a variability language enables the creation of editor views for specific configurations.

R3 (Validation). We use default configurations to execute the new generator on existing ADS code in order to ensure that it produces the same output as the legacy generator.

To increase the test coverage, we currently experiment with symbolic execution [4] on the preprocessor language to identify test inputs that trigger more program paths.

For the symbolic analysis, we declare the parameters of a generator as *symbolic*. The analysis searches feasible paths through the generator and provides corresponding *concrete* parameter values which will trigger this path. These values are the test input we run on both the legacy and the modernized tool and compare their results. Unfortunately, symbolic execution tools only exist for common programming languages like C. To build a custom analysis for the preprocessor language, we have to address specific challenges, such as dynamic typing and the strong reliance on string operations.

Currently, we have an incomplete implementation that, however, can already provide valid test inputs for many generators. We can measure the test coverage achieved by the generated test inputs and use this as a benchmark for the analysis.

R5 (Editor). With the use of different DSLs to implement the sub-generators, it is easy to create individual syntax highlighting and other editor features for each DSL.

The projectional editor in MPS requires a different interaction with the IDE than developers are used to, which may challenge acceptance of such a solution. Existing studies show that developers can use a projectional editor at least as efficient as a normal editor after an initial training phase [3, 19]. As a consequence, the migration will need to be complemented by sufficient training for developers.

R6 (Language Extensions). MPS enables even non-experts to make changes to the languages. Any change in the semantics of the preprocessor language will require a corresponding change in the preprocessor, which is not very complicated. For example, during the case study an undergraduate student developed parts of the preprocessor transformation.

Any addition to the language—for example adding logical operators—will only be useful if they are actually used. Developers could use new features for newly developed code or when refactoring old code. Additionally, we can automatically refactor the old codebase during import to make use of the new features. For example, we implemented an analysis to identify places where nested `if` statements can be merged into a single condition combined with `&&`.

Making changes to the new generator—for instance, to increase readability—is also easy. For example, the sub-generator SPP generates loop functionality using `goto` statements, which used to increase compatibility with older com-

```
>> 01     STATUS-PROGRAM PIC #01.;
>> 01     DCV-ERROR-SEVERITY PIC #01.;
.IF ( #SA . CO . NODELTAITS )
  >> 01     DCV-SC-NEXT     PIC #01 VALUE ZERO.;
.ELSE
```

Figure 6: Added symbols (>> and ;) to support editing in the projectional editor

pilers. With a few changes in the *TextGen* aspect we modernized the generated output to emit the same functionality with a native COBOL loop statement, which nowadays supported by every compiler in use.

R7 (Documentation). We created a simple generator for HTML documentation. The resulting documentation is not really useful yet, because it only contains a few pieces of information from the nodes, but it demonstrates that it is easy to generate documentation.

R8 (Modularization). Our solution is modular, because the features (preprocessing and sub-generators) of the legacy generator are mapped to individual languages. Each language can be developed independently.

The key advantage of a projectional editor is the ability to compose DSLs—we compose the preprocessor language and the languages for the sub-generators. An obvious extension is to implement the target language, which is currently COBOL as a DSL in MPS as well. This will enable much more support in the editor, including syntax highlighting and auto-completion, for the embedded target language fragments.

R9 (Target Language). In the long term, a change of the embedded target language could be fully based on MPS, which provides dedicated tools for language migrations. To this end, both the current and the future target language need to be DSLs in MPS, which in our case would require designing the language in MPS and building an importer. Unlike the ADS languages we migrated, COBOL is a more complex general-purpose language whose migration will be more expensive.

6. LESSONS LEARNED

We synthesize our *lessons learned* to support practitioners who plan a similar migration of a language infrastructure.

Trade-off between Preserving Syntax and Using a Projectional Editor. The projectional editor in MPS allows for much flexibility in the language syntax. As a start, we targeted to recreate the syntax from the legacy language. This approach makes it easy to visually compare the original code with the new code to manually validate the import step. Furthermore, it helps developers of legacy systems to cope with the change of the generator.

However, the resulting syntax is not necessarily optimal for use with the projectional editor. For example, we had to cope with a case of ambiguity in the user interaction: we defined code of the embedded target language to be a list of simple text pieces (COBOL code) or a reference to a variable. However, this way it could not be distinguished whether the user intents to add a new item or to create a line break, when she presses Enter with the cursor at the end of such a list. Fig. 6 shows how we mitigated this ambiguity by introducing a semicolon as the end-of-line character, which did not exist in the legacy language. Now, with a cursor positioned *before* the semicolon a user will add new items to the list. A cursor position *after* the semicolon allows the creation of the elements after the list.

A known problem with model-based editing is that users can no longer use custom formatting of their source code [5]. In our case, the preprocessor language contains decision tables as a special control structure, which were often formatted using white spaces. This formatting will be lost during import: instead we need to explicitly model the indented layout, such as using tables in the editor.

Create MPS-independent Libraries. To ease and decouple the re-implementation of library code that is not directly part of the generator language, it turned out useful to develop some components independently from MPS and just integrate them as a JAR library.

Use MPS to Create Debuggers. MPS provides an API to create custom debuggers for a DSL. The idea is to build a debugger with a front-end showing the DSL and a back-end connected to a standard debugger. This approach is used by MPS itself, which uses the Java debugger in the backend, and by the mbeddr project, which connects to gdb for debugging [14]. We evaluated the debugger API for our preprocessor with a small prototype but have yet to build a working debugger for the generator DSLs.

The execution of MPS itself, including our importer and the transformations, can be debugged with a breakpoint debugger in MPS. To debug the model-to-model generation, MPS provides two instruments. First, the *generation plan* shows the order in which model transformations are executed. Second, *transient models* can be used to show intermediate steps in the model transformation.

Limitations. The original implementation of the preprocessor language is purely text-based, similar to CPP, which therefore supports both disciplined and undisciplined annotations [12]. Our solution only supports disciplined annotations. That is, the preprocessor is integrated with the AST—among others, the two different branches of a preprocessor `IF` statement need to be of the same AST type. CPP does not have this limitation, and `IF` statements can cross-cut any text token of the target language. However, in our sample of AST code we used to validate the migration, undisciplined annotations were rare, and in fact our solution, which prevents undisciplined annotations, led to a better DSL design.

7. RELATED WORK

Related work comprises existing case studies on migrating language infrastructures and approaches to improve the handling of variability annotations.

Voelter et al. present a case study of a forward-engineering project using mbeddr [16]. Our case study has a re-engineering background, we therefore focus on the migration to a language workbench. Recently, Méndez-Acuña et al. introduced PUZZLE to enable reuse between similar legacy DSLs, which in turn should reduce the effort necessary to implement DSLs [13]. While their approach assumes specifications of existing DSLs, we started by decomposing an existing system into DSLs.

Many works on migrating COBOL systems to a modern platform exist, including tools and experience reports. Some also consider the presence of COBOL generators. Fleurey et al. present a model-driven software migration approach and a case study on migrating a banking application using

COOL:Gen (another COBOL generator) to Java [7]. Similar to our approach, they define a target model and corresponding transformations to build a scalable migration. Unlike our approach, they remove the generator and directly modernize the application, which would not fit our requirements on improving the variability handling and the incremental modernization.

Another idea is to base the migration on the generated COBOL code [2], evading the need to re-implement the generator language. This approach has the disadvantage that such a migration will be only valid for a single configuration—the code will no longer contain any variability. Furthermore, generated code is harder to understand, since the generator's abstractions are lost, as well as the original design of the developers and additional information, such as comments.

Finally, there are multiple approaches to move legacy systems to a software product line approach [11]. Our case study differs from these, since we do not just focus on the migration of a single legacy system, but on the migration of the language infrastructure to handle variability.

8. CONCLUSION

We presented a case study of migrating the legacy generator tool ADS and a substantial codebase of ADS code to a modern language workbench. Our focus was on ADS' preprocessor sub-language, mainly used to realize variants for individual customers. The case study showed the feasibility of conducting such a migration. The main benefits are an increased maintainability and extensibility of the generator languages in order to support future changes in the (sub-)languages, but also modern editor support. MPS' projectional editor even allows having different concrete syntaxes (which could also include graphical elements, such as tables), which does not require changing the abstract syntax or the existing code, which is always saved as a model in abstract syntax. One of the major strengths of a projectional editor is the support for integrating sub-languages with overlapping keywords, which parsers could not easily disambiguate. This strength was not necessary (since the decomposed languages were already integrated before), but could also allow future integrations of languages with ambiguous concrete syntax.

Acknowledgments

Lillack's work is supported by the German Federal Ministry of Education and Research under grant number 01IS15009B.

9. REFERENCES

[1] ADM Task Force. Architecture-driven modernization scenarios. Technical report, OMG, 2006.

[2] F. Barbier, G. Deltombe, O. Parisy, and K. Youbi. Model driven reverse engineering: Increasing legacy technology independence. In *IWRE*. 2011.

[3] T. Berger, M. Voelter, H. P. Jensen, T. Dangprasert, and J. Siegmund. Efficiency of projectional editing: A controlled experiment. In *FSE*, 2016.

[4] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[5] S. Erdweg, T. van der Storm, M. Voelter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages Systems & Structures*, 44 Part A:24 – 47, 2015.

[6] 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 *SLE*, 2013.

[7] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jézéquel. Model-driven engineering for software migration in a large industrial context. In *MODELS*, 2007.

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

[9] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *OOPSLA*, 2011.

[10] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage. How do professionals perceive legacy systems and software modernization? In *ICSE*, 2014.

[11] M. A. Laguna and Y. Crespo. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Sci. Comput. Program.*, 78(8):1010–1034, Aug. 2013.

[12] J. Liebig, C. Kästner, and S. Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *AOSD*, 2011.

[13] D. Méndez-Acuña, J. A. Galindo, B. Combemale, A. Blouin, B. Baudry, and G. Le Guernic. Reverse-engineering reusable language modules from legacy domain-specific languages. In *ICSR*, 2016.

[14] D. Pavletic, M. Voelter, S. A. Raza, B. Kolb, and T. Kehrer. Extensible debugger framework for extensible languages. In *Ada-Europe*, 2015.

[15] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[16] M. Voelter, A. v. Deursen, B. Kolb, and S. Eberle. Using C language extensions for developing embedded software: A case study. In *OOPSLA*, 2015.

[17] M. Voelter and V. Pech. Language modularity with the MPS language workbench. In *ICSE*, 2012.

[18] M. Voelter, D. Ratiu, B. Kolb, and B. Schätz. mbeddr: instantiating a language workbench in the embedded software domain. *Autom. Softw. Eng.*, 20(3):339–390, 2013.

[19] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. Towards user-friendly projectional editors. In *SLE*, 2014.