Towards User-Friendly Projectional Editors

Markus Voelter[1], Janet Siegmund[2], Thorsten Berger[3], and Bernd Kolb[4]

[1] independent/itemis, `voelter@acm.org`
[2] Universität Passau, `Janet.Siegmund@uni-passau.de`
[3] University of Waterloo, `tberger@gsd.uwaterloo.ca`
[4] itemis AG, `kolb@itemis.de`

Abstract. Today's challenges for language development include language extension and composition, as well as the use of diverse notations. A promising approach is projectional editing, a technique to directly manipulate the abstract syntax tree of a program, without relying on parsers. Its potential lies in the ability to combine diverse notational styles – such as text, symbols, tables, and graphics – and the support for a wide range of composition techniques. However, projectional editing is often perceived as problematic for developers. Expressed drawbacks include the unfamiliar editing experience and challenges in the integration with existing infrastructure. In this paper we investigate the usability of projectional editors. We systematically identify usability issues resulting from the architecture. We use JetBrains Meta Programming System (MPS) as a case study. The case study discusses the concepts that MPS incorporates to address the identified issues, evaluates effectiveness of these concepts by surveying professional developers, and reports industrial experiences from realizing large-scale systems. Our results show that the benefits of flexible language composition and diverse notations come at the cost of serious usability issues – which, however, can be effectively mitigated with facilities that emulate editing experience of parser-based editors.

1 Introduction

As expressed by *closeness of mapping* in the cognitive dimensions of notations [1], the degree to which we can effectively express facts in a given domain is heavily influenced by the alignment of the used language with that domain. This applies to programming languages, but also to domain-specific languages (DSLs) used in a wide range of technical and business domains. However, a language must also use a suitable notation. Imagine mathematics represented as a linear sequence of characters, without integral symbols, fraction bars or superscripts: it would be much harder to read, making mathematics as a language less useful – more *hard mental operations* [1] would be due to the syntax and not the underlying semantics. DSLs are often targeted at non-programmers. While the suitability of a language for its target audience is guided by many criteria (as discussed in [1]), our experience tells us that that the notation is especially important for languages targeted at non-programmers. Another important concern in languages is their composability (approximated by *juxtaposability* in [1]). Software systems are often expressed with a set of languages (some used by programmers, some by other stakeholders), and these languages must be integrated in terms of syntax, semantics, and their development environments: today, IDEs are essential to languages since users increasingly rely on IDEs to efficiently edit programs.

Traditionally, languages use either textual or graphical notations. Each kind of notation comes with its own editor architecture. Textual notations are typically edited with text buffers, grammars and parsers. The supported notations are essentially linear sequences of characters and – depending on the grammar class – in their ability to compose independently developed languages. Graphical notations use direct manipulation instead of parsers. But purely graphical notations are only suitable for a limited set of languages, and many real-world languages require a mix of graphical, textual, tabular and symbolic/mathematical notations. Projectional editors (ProjEs) support this approach. They generalize the approach used in graphical editors to arbitrary notations. Editing gestures directly change the abstract syntax tree (AST). Users see and interact with a rendering of the AST called a projection. There is no transformation (that is, parsing) from the concrete syntax to the AST. This allows non-textual notations, as demonstrated by intentional programming [2,3], which relies on projectional editing. ProjEs also avoid the problems with compositionality known from grammar-based systems: ambiguities cannot arise since no grammars are used.

However, ProjEs have traditionally had two problems. First, for notations that look textual, users expect that the editing behavior resembles classical text editing as much as possible. Historically, ProjEs have not been good at this; users had to be aware of the AST when editing programs, leading to usability problems. For example, when entering `2+3`, users first had to enter the `+` and then enter the two arguments. Second, ProjEs cannot store programs in the concrete syntax – otherwise, this syntax would have to be parsed when programs are loaded into the editor. Instead, programs are stored as a serialized AST, often as XML. This makes the integration with existing infrastructures, such as version control systems (VCS) or diff/merge tools, a challenge.

Hypothesis Although ProjEs have been around for a long time (see Section 2.2), and despite their demonstrated advantages in terms of notational flexibility and support for language composition and extension, ProjE have not seen much adoption in practice. We hypothesize that this is mainly because of the drawbacks regarding editor usability and infrastructure integration discussed above.

Goals, Methods, and Contributions Our goal is to evaluate the usability of projectional editors. To this end, we first systematically identify and categorize usability issues arising from the architectural peculiarities of projectional editors. We then provide a case study of a state-of-the-art projectional editor – the JetBrains Meta Programming System (MPS). In the case study, we discuss the techniques used by MPS to mitigate the identified issues, and evaluate their effectiveness by surveying professional developers. We finally report industrial experiences from realizing large-scale systems. We contribute: (i) a taxonomy of usability issues that projectional editors face, (ii) a mapping of concrete mitigation techniques for the issues, and (iii) empirical data on how professional developers perceive effectiveness of projectional editing.

Results We identify 14 usability issues related to efficiently entering code (e.g., non-linear typing), selection and modification of code (e.g., introducing cross-tree parentheses), and integration with existing infrastructure (e.g., version control systems). Half of these issues can be addressed sufficiently, for instance, using code completion or expression-tree-refactoring support. Others require language- or notation-specific implementations, or cannot be mitigated conceptually. Results of the survey show that developers perceive projectional editing as an efficient technique applicable in every-day work, while the effort of getting used to it is high. However, the survey also reveals weaknesses, such as the support for commenting, which is currently not addressed sufficiently in MPS.

2 Background

2.1 Parsing vs. Projection

In parser-based editors (ParEs), users type characters into a text buffer. The buffer is then parsed to check whether a sequence of characters conforms to a grammar. The parser builds a parse tree, and ultimately, an abstract syntax tree (AST), which contains the relevant structure of the program, but omits syntactic details. Subsequent processing (such as linking, type checks, and transformation) is based on the AST. Modern IDEs (re-)parse the concrete syntax while the user edits the code, maintaining an up-to-date AST in the background that reflects the code in the editor's text buffer. However, even in this case, this AST is created by a parser-driven transformation from the source text.

A ProjE does not rely on parsers. As a user edits a program, the AST is modified *directly*. A projection engine uses projection rules to create a representation of the AST with which the user interacts, and which reflects the resulting changes. No parser-based transformation from concrete to abstract syntax involved here. Fig. 1 shows the difference. This approach is well-known from graphical editors: when editing a UML diagram, users do not draw pixels onto a canvas, and a "pixel parser" then creates the AST. Rather, the editor creates an instance of `uml.Class` when a user drops a class onto the canvas. A projection engine renders the diagram by drawing a rectangle for the class. Programs are stored using a generic tree persistence format (such as XML). As the user edits the program, program nodes are created as instances of language concepts. This approach can be generalized to work with any notation, including textual. A code-completion menu lets users create instances based on a text string entered in the editor called the *alias*. The concepts available for instantiation (and, thus, the valid text strings/aliases) depend on the language definition. Importantly, *every next text string is recognized as it is entered*, so there is never any parsing

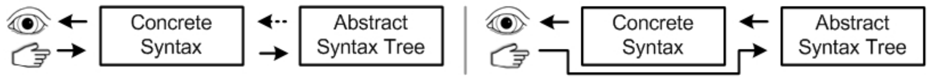

Fig. 1. In ParEs (left), users see and modify the concrete syntax. A parser constructs the AST. In ProjEs, users see and interact with the concrete syntax, but changes *directly* affect the AST. The concrete syntax is projected from the changing AST.

of a sequence of text strings. In contrast to ParEs, where disambiguation is performed by the parser after a (potentially) complete program has been entered, in ProjEs, disambiguation is performed by the user as he selects a concept from the code-completion menu. Once a node is created, it is *never* ambiguous what it represents, *irrespective of its syntax*: every node points to its defining concept. Every program node has a unique ID, and references between program elements are represented as references to the ID. These references are established during program editing by directly selecting reference targets from the code-completion menu; the references are persistent. This is in contrast to ParEs, where a reference is expressed as a string in the source text, and a separate name resolution phase resolves the target AST element after the text has been parsed.

2.2 Related Work in Projectional Editing

An early example of a ProjE is the Incremental Programming Environment (IPE) [4]. It supports the definition of several notations for a language as well as partial projections, where parts of the AST are not shown. However, IPE suffers from the problem with editing expressions introduced earlier: to enter `2+3`, users first have to enter the `+` and then fill in the two arguments. This is tedious and forces users to be aware of the language structure at all times. IPE also does not address language modularity; it comes with a fixed, C-like language and does not have a built-in facility for defining new languages. Another early example is GANDALF [5], which generates a ProjE from a language specification. Even though [6] does not report on a systematic study, the authors expect the same usability problems as IPE: "Program editing will be considerably slower than normal keyboard entry, although actual time spent programming non-trivial programs should be reduced due to reduced error rates." The Synthesizer Generator [7] is also a ProjE. However, at the fine-grained expression level, textual input and parsing is used. While this improves usability, it destroys many of the advantages of projectional editing in the first place, because language composition *at the expression level* is limited. In fact, extension of expressions is particularly important to tightly integrate an embedded language with its host language [8].

The Intentional Programming [2,3] project has gained widespread visibility and has popularized projectional editing; the Intentional Domain Workbench (IDW) is the contemporary implementation of the approach. IDW supports diverse notations [9,10]. However, we are not aware of any studies regarding its usability, and since it is a commercial system, we cannot evaluate it. Our understanding is that the IDW has not found widespread adoption so far.

Language boxes [11] rely on explicitly delineating the boundaries between different languages used in a single program (e.g., the user could change the box with `Ctrl-Space`). Each language box may use parsing or projection. This way, textual notations can be edited naturally, solving the usability issues associated with editing text in a ProjE. However, it is not clear whether fine-grained mixing between different boxes will work in terms of usability. For example, consider a projectional editor for a mathematical notation embedded (in its own box) inside an otherwise textual editor for C code. As part of the mathematical expression, users would like to use (textual) references to C variables. Providing an integrated user experience, as well as integrated symbol tables, may not be a trivial problem. In addition, language boxes address *only* the usability problem: the approach still requires a specialized IDE (that knows about the boxes) plus non-concrete syntax storage (because the boxes must be represented somehow).

Hybrid editors are another alternative of solving the usability problems of ProjEs by on-demand parsing. Unlike in a ParE, the editor content consists of atomic tokens, not characters. These tokens have normal projectional editors. This makes it possible to embed complex tokens, such as diagrams or math symbols, and still edit sequences of such tokens linearly. A prototype is currently being explored by a team at JetBrains (available at `http://jb-proj-demo.appspot.com/index.html`). It is not clear at this point what the trade-offs are regarding language composability, notational freedom, and usability.

2.3 Case Study: MPS and mbeddr

JetBrains MPS (`http://jetbrains.com/mps`) is an open-source language workbench that uses projectional editing. It is a comprehensive environment for language engineering, supporting language aspects such as concrete and abstract syntax, type systems and transformations, as well as IDE aspects, such as syntax highlighting, code-completion, find-usages, diff and merge, refactoring, and debugging. It also supports language modularization and composition [8].

We have chosen MPS as our case study for three reasons. (1) MPS is currently the most widely used ProjE. It is used for various projects, including JetBrains YouTrack, mbeddr (discussed below), computational biology [12], web applications (`http://codeorchestra.com/ide/`), requirements engineering [13], and insurance DSLs. (2) Some of the authors of this paper have significant industry experience with MPS. (3) MPS is open-source, which fosters replicability of our results.

MPS relies on a meta meta model very similar to EMOF and EMF Ecore [14]. Language concepts (corresponding to meta classes) declare children (single or lists), references and primitive properties. Concepts can extend other concepts or implement concept interfaces. Subconcepts can be used where a superconcept is expected (polymorphism). Programs are represented as instances of concepts, called nodes. Each concept also defines one or more editors. These are the projection rules that determine the notation of instance nodes in the program. The editor also defines intentions, little in-place program transformations that can be triggered by the user as he edits the program.

mbeddr (`http://mbeddr.com`) is an extensible set of integrated languages for embedded software engineering [15], developed with MPS. mbeddr is also open source. It is primarily used for implementing embedded systems, ranging from relatively small examples (such as Lego Mindstorm robots) to non-trivial commercial applications (e.g., a smart meter [16]). mbeddr has been chosen by Siemens PLM Software (formerly LMS) as the basis of a new controls engineering tool, which is currently being developed as a set of mbeddr extensions.

The core of mbeddr is an extensible version of C99 and a set of extensions for embedded software, such as interfaces and components, state machines or physical units. mbeddr provides multi-paradigm programming for C [17], in which different abstractions can be used and mixed in the same program. mbeddr also supports languages for cross-cutting concerns, such as documentation, requirements management, and traceability, as well as product-line engineering. Several formal verification techniques are also directly integrated with the languages.

3 Advantages and Drawbacks of Projectional Editing

We now systematically analyze the usability challenges traditionally associated with ProjEs. We have identified three categories: efficiently entering (textual) code (EE), selecting and modifying code (SM), as well as infrastructure integration (II). These categories reflect anecdotal evidence on usability challenges of ProjEs. They are also obviously relevant for productively using an editor. For each of the categories, we identify and explain specific challenges in the following sections.

3.1 Efficiently Entering (Textual) Code

Most grammars used in practice by ParEs are not freely composable, because the composed grammar may become ambiguous. The details depend on the grammar class used by the parser, and various disambiguation approaches are used to address the issue. We mention two examples below; an extensive discussion can be found in [8]. Formalisms that implement full context-free grammars compose much better, depending on the modularity of the grammar language [18]. An example of a grammar formalism that supports only limited composition is ANTLR [19]. In contrast, SDF2 [20] supports full context-free grammars based on a scannerless generalized LR parser, and composition support is much better: As an example, [21] demonstrates embedding SQL into Java. Disambiguation is necessary if the same syntactic form is used in the same location to represent different language concepts (i.e., must be parsed differently). SDF2 performs disambiguation via quotations, and SILVER/COPPER [22] uses disambiguation functions. In ProjEs, since no grammars are used, language composition is unlimited (discussed systematically in [8]). Situations which would lead to an ambiguity in ParEs are resolved by **asking the user to manually disambiguate (EE.1)** at the time of entering the potentially ambiguous code. As an example of composition and extension, the mbeddr system currently has over 30 modular extensions to C; all of them can be used in the same program. Many of them are illustrated in [15].

The manual disambiguation also includes references: Targets are picked from the code-completion menu. This means that users **cannot establish references to non-existing nodes (EE.2)**, because, if they do not exist yet, the code-completion menu cannot offer them to the user. In ParEs, a user can just type `i++` even though `i` has not yet been declared. The user can go back later, and add a declaration of `i` before its use. This works because the `i` in `i++` is just a symbol, and its resolution happens later – it is marked as an error as long as no declaration for `i` is in scope. In a ProjE, every reference is an actual pointer to its target. If the target does not exist, the reference cannot be entered.

Textual projections require the AST to be projected linearly. As discussed in Section 2.2, ProjEs have traditionally forced the tree structure on the user even when the notation was linear, i.e., they **required structure-aware typing (EE.3)**. `2+3` must be entered by first typing the `+` and then entering its two children, instead of just linearly typing `2+3`.

ParEs extract structure from characters in a text buffer based on a grammar. Mainstream grammars work on linear sequences of characters. This severely limits non-linear notations, such as math/symbols (because they are two-dimensional) or graphics, and limits tabular notations to simple cases where the vertical bar (`|`) is used to separate columns and rows (as shown by Jnario (`http://jnario.org`), a language for behavior-driven development) or simple, non-recursive fraction bar-like notations (used for type system rules in [23]). Coordinate grammars [24] have been proposed to parse two-dimensional mathematics structures. Parsers for visual notations have been proposed as well; for example, [25] discusses parsing of hand-inputted shapes on tablet computers based on a formalism called set grammars. More general discussions on parsing visual languages are provided in [26] and [27]. However, these grammars use different formalisms and so do not easily integrate with traditional grammars for linear text. None of these approaches has found its way into industry-strength language tooling.

Since ProjEs never parse the concrete syntax, they can use notations that are not parseable, or use two-dimensional layout. Examples include tables, mathematical symbols (fraction bars, superscript or $\sum$) or diagrams. This is discussed for IDW in[9,10] and for MPS/mbeddr in [15]. ProjEs can also mix different notational styles. For example, tables can contain textual expressions and mathematical symbols (as in mbeddr's decision tables), and textual programs can embed graphics. This works because all notational styles are implemented using the same projectional architecture. In contrast, maintaining an integrated overall model created with editors that combine parsing and diagram editing is challenging for a number of technical reasons [28]. These include that parser-based editors use (qualified) names to represent program node identity, whereas graphical editors natively use IDs for this purpose or that references in ParEs are created via name binding, and graphical editors use the unique ID.

This notational flexibility leads to drawbacks. In a ParE, a program can always be typed exactly the way it looks by typing the sequence of characters one by one. In a ProjE, it is possible to project program nodes in arbitrary ways, including tables or mathematical symbols; these cannot just be typed. For example, the $\sum$ is not available on the keyboard. Thus, it cannot be deduced from just looking at a program (e.g., in a presentation or a book) how to enter it: **What you see is not what you type (EE.4)**. More generally, the different notational styles may **require notation-specific editor support (EE.5)**, each potentially with their own idiosyncrasies.

Many ProjEs support the definition of multiple editable notations for the same language structure. A program's representation can be switched on the fly by selecting another set of projection rules. This is not practical for ParEs, since most useful changes in representation also lead to changes in the underlying structure. As an example, mbeddr supports editing state machines either as text, or as tables, and a graphical notation is currently being added. Also, in a ProjE, a program can contain data that is not shown in the projection, and partial projections or views are possible. This is because the program is stored as the AST, which contains all data, even when it is not shown. For example, mbeddr stores requirements traces [29] in programs. In contrast, a ParE must always contain all data in the concrete syntax, because this is the persistent representation. It is possible to hide some parts, but this requires specific, language-aware support in the editor. The Jnario editor can optionally hide the formal aspects of tests. However, this flexibility means that programs cannot be stored in their concrete syntax, requiring persisting programs as a generic tree structure. This leads to challenges with infrastructure integration (discussed below).

3.2 Selecting and Modifying Code (SM)

In ParEs, selection happens in the text buffer: any character, word, line (or sequences thereof) can be selected and subsequently changed, cut, copied or pasted. In a ProjE, **selection is based on the tree structure (SM.1)**: nodes, parent nodes, or siblings in lists can be selected. This also makes it **hard to perform cross-tree modifications (SM.2)**, i.e., editing structures that are not aligned with the tree. Consider the expression `1 + 2*3`. To change this into `(1+2) * 3`, parentheses have to be inserted in places that cross-cut the tree structure: most ProjEs do not support this, and the expression has to be retyped. Finally, **copy and paste is structure-aware (SM.3)**, and not just based on the syntax. If a user wants to paste something in a location where it may fit syntactically, but the underlying AST uses a different language concept, this will not work. An example is pasting a C `Function` into a C++ class, where it needs to be a `Method` instead, even though it has essentially the same syntax.

In ParEs, it is sometimes hard to detect semantic associations between program elements, since such associations are expressed by "geographical proximity". For example, comments are typically located above the program element they belong to. In a ProjE, the relationship between program nodes is typically designed to be explicit: for example, comments would be children of the element they are associated to, even though they may still be projected above it. This results in more robust merging and refactoring, but also means that a ProjE has **no support for free-floating comments (SM.4)**.

In a ParE, code that is temporarily not needed can be commented out. It is then ignored by the compiler, type checker, and other IDE services; it is treated as plain text. When the code is needed again, it can be uncommented: the parser parses the text and (re-)creates the AST. In a ProjE, commenting is not so easy, since the commented code must retain its structure so it can be uncommented later when it is needed again. Hence, **dedicated support for commenting code is required (SM.5)**.

Whitespace is typically ignored by ParEs, and not explicitly described in grammars. To be able to pretty-print a program after an automated modification, an additional pretty-printing specification is typically required. In a ProjE, this is not required, since the projection rules already contain layout information.

		Issue	Mitigation Technique used by MPS
Efficiently Entering (Textual) Code	EE.1	Requires manual, user-based disambiguation	code completion, aliases, context constraints
	EE.2	Cannot establish references to non-existing nodes	intentions to create missing targets
	EE.3	Requires structure-aware typing	side transforms, delete actions, smart references, wrappers, smart delimiters
	EE.4	What you see is not what you type	–
	EE.5	Requires notation-specific editor support	– (but editors share common aspects)
Selecting and Modifying Code	SM.1	Selection is based on the tree structure	–
	SM.2	Hard to perform cross-tree modifications	expression tree restructuring
	SM.3	Requires structure-aware copy/paste	paste handlers
	SM.4	Does not support free-floating comments	– (partly addressed by metamodel extension in mbeddr)
	SM.5	Requires dedicated support for commenting code	– (partly addressed by metamodel extension in mbeddr)
	SM.6	Does not support custom layout	–
Infrastructure Integration	II.1	Requires tool support for diff/merge	node-by-node revert, merge driver, diff/merge tool using projection rules
	II.2	Text-based shell-scripting tools cannot be used	– (build system support for generating and testing models)
	II.3	Requires tool support to export/import textual syntax	copy/paste, parser hooks, generic node (de-)serialization

Table 1. Mapping of identified usability issues to mitigation techniques

On the flip side, a ProjE **does not support custom layout (SM.6)** – the representation is determined completely by the projection rules.

3.3 Infrastructure Integration (II)

Today's development infrastructure is geared towards text files, and ParEs integrate seamlessly. The diff/merge facilities of VCS rely on showing the file contents. This works well for concrete syntax storage, but it does not work for AST-based storage. Special **tool support for diff/merge is needed (II.1)**.

Tools such as `grep` assume concrete syntax storage. While ProjEs store names as strings (so they can be `grep`'ed), more complex structures are represented as several nodes and `grep`'ing for their concrete syntax representation will not work. **Text-based shell scripting tools cannot be used (II.2)**. A ProjE will typically support searching on the projected syntax, but the ProjE must be used for the purpose; generic text-search tools are not enough.

Code written in a ParE can trivially be pasted to and copied from another text-based application. For a ProjE, this is not necessarily so simple; **tool support is required to export/import textual syntax (II.3)**. Non-textual notations, such as tables or symbols, cannot be pasted to a text editor at all.

4 Addressing the Drawbacks in Projectional Editors

In this section, we revisit the problems associated with ProjEs introduced in Section 3 and illustrate the mechanisms (typeset in *italic*) used by MPS to address them. Some of these approaches have already been introduced in [15]. Table 1 summarizes the issues and MPS' mitigation techniques where applicable.

4.1 Efficiently Entering (Textual) Code (EE)

EE.1 Requires manual, user-based disambiguation Disambiguation in MPS relies on the user selecting the correct language concept from the *code-completion* menu, whose contents are driven by the language structure. Language concepts define an *alias*, the string used to pick the concept from the code-completion menu. By making the alias the same as the leading keyword (e.g. `if` for an `IfStatement`), users can "just type" the code. MPS also supports *context constraints* that restrict the locations where concepts can be used based on arbitrary conditions. For example, mbeddr has different `assert` keywords, each with different translation to C. To avoid confusing the user by requiring manual disambiguation between them, context constraints ensure that each of these `assert` statements is available only in disjoint contexts.

EE.2: Cannot establish references to non-existing nodes MPS supports *intentions to create the missing targets* in a context-dependent way. For example, if a user enters a global variable in mbeddr C as `int32 global = someName`, where `someName` does not exist, MPS provides two intentions: one to create a global variable `someName`, and one to create a global constant. If a user enters a local variable (in a function) as `int32 local = someName`, there are two more intentions that support creation of a local variable and a function argument.

EE.3: Requires structure-aware typing Consider an expression `2` that should be changed to `2 + 3`. MPS supports *side transforms* to allow users to simply type `+` on the right side of the `2`. The transform replaces the `2` with the `+`, puts the `2` in the left slot, and then puts the cursor into the right slot so the user can enter the second argument. Side transforms also reshuffle the tree to ensure it reflects operator precedence: higher precedence means the operator is further down in the tree. Precedence is typically specified by a number associated with each operator. *Delete actions* are used for a similar effect when elements are deleted. Pressing `Backspace` on the `3` in `2 + 3` keeps the `2 +`, with an empty right slot. Pressing `Backspace` on the `+` replaces it with its left argument, the `2`.

References are also established via code-completion. Consider pressing `Ctrl-Space` after the `+` in `2 + 3`. If local variables are in scope, these should be available in the code-completion menu. However, technically, a `VarRef` has to be instantiated first, whose `variable` slot is then made to point to a variable. This is tedious, and *smart references* solve the problem: If a `VarRef` is allowed in a given context, the editor first finds the possible targets and puts those targets into the code-completion menu. Only after the user has selected a target, then the `VarRef` is created, and the selected element is put into its `variable` slot. This makes the reference object invisible in terms of the editing experience.

Consider a local variable declaration `int a;` represented by the concept `LocalVarDecl`, a subconcept of `Statement` so it can be used in function bodies. Users expect to be able to enter a local variable by typing `int`. However, `int` is a `Type`, and a `Type` is not legal in a statements list – a statement list expects instances of `Statement` – and hence cannot be entered. *Wrappers* solve this problem: if a `Type` is entered in `Statement` context, the wrapper creates a `LocalVarDecl`, puts the `Type` into its `type` slot, and moves the cursor into the `name` slot. This way, a local variable declaration `int a;` can be entered by starting to type the `int` type, as expected.

Finally, *smart delimiters* are used to simplify inputting lists that are separated with a separator symbol (such as the arguments in a function). Typing the separator (e.g., comma), automatically adds a new node to the list.

EE.4: What you see is not what you type The problem that some concepts (such as $\sum$) cannot be entered just by typing what is projected cannot be solved; it is a consequence of allowing notations that are not on the keyboard.

EE.5: Requires notation-specific editor support The editors used for the different notations share common aspects: code completion and intentions work everywhere, selection is always based on the tree structure, and pressing `Backspace` on a program element always deletes it. Still, notation-specific gestures have to be learned. For instance, the table editors offer special gestures to create new rows, and graphical editors require the mouse to move elements.

4.2 Selecting and Modifying Code (SM)

SM.1: Selection is based on the tree structure MPS provides no solution to this problem. `Ctrl-Up/Down` selects along the tree structure. `Shift-Up/Down` selects siblings in child lists. This works independent of the notations. For example, if a tree is projected as a table, `Ctrl-Up` will select the current row if that row represents the parent node, and then `Shift-Down` selects rows under the current one if the corresponding nodes are siblings. As this example illustrates, selection based on the tree structure is not always bad, because programs are highly structured. This is also illustrated by the fact that some ParEs (such as Eclipse) support tree-based selection in addition to character-based selection.

SM.2: Hard to perform cross-tree modifications Cross-tree editing, as in changing `1 + 2*3` to `(1+2) * 3` is solved as follows: a separate opening parenthesis can be entered anywhere in the tree, and its position is remembered temporarily. Upon entering a corresponding closing parenthesis, the *expression tree is restructured* to reflect the new structure indicated by the inserted parentheses.

SM.3: Requires structure-aware copy/paste To address the problem of not being able to paste an instance of concept `A` in a program location where an instance of `B` is expected, MPS supports *paste handlers*. These are callbacks that transform an instance of `B` to an instance of `A` if the paste context requires it.

SM.4: No support for free-floating comments Free-floating comments remain unsupported in MPS. mbeddr supports attaching comments to all program elements that implement an interface `IDocumentable`. All top-level mbeddr C constructs and all statements implement this interface, so essentially everything except expressions or types can be commented. In addition, mbeddr support a `CommentStatement`, which means that procedural code (such as function bodies) can contain comments that are not associated with any particular element.

SM.5: Requires dedicated support for commenting code Unfortunately, MPS provides no generic support for (temporarily) commenting out code. mbeddr uses the following approach: If instances of some concept should be commentable, a subconcept is defined that implements an interface `ICommentedCode`. The subconcept stores the commented code and is marked to suppress errors. It also overrides the editor styles to use a uniformly gray text color. Using this approach, it is relatively simple to make statements or module contents commentable. However, the approach does not work for commenting out parts of expressions, as in `1 + 2 * (4 /*+7*/)`.

SM.6: Does not support custom layout MPS does not support user-defined layout. However, the projection rules can be defined with conditional projections so that, for example, a statement list that contains only one statement is rendered on one line (as in `if (..) { return x; }`) instead of over several lines. Conditional projections can also be used to implement user-definable preferences, such as whether the opening curly brace should be on a new line or not. More generally, it is not clear whether predefined layout is actually a problem: many organizations mandate formatters that enforce a predefined layout.

4.3 Infrastructure Integration (II)

II.1: Requires tool support for diff/merge The fact that MPS stores its models in XML files (and not in a database) means that MPS can be integrated with file-based development infrastructures. More specifically, the VCS integration involves the following ingredients. First, the editor highlights those parts of programs that have changed since the last update, shows diffs of these parts, and supports reverting changes on a *node-by-node basis*. Second, while diff/merge is performed by the underlying VCS, MPS ships with a *merge driver* that makes sure the merging process respects the idiosyncrasies of MPS' XML format. Finally, any diff or merge that requires manual user intervention is performed in an internal *diff/merge tool that uses the projection rules*. It works for any notation, and for textual languages, diff/merge works exactly as in text-based merge tools.

II.2: Text-based shell-scripting tools cannot be used The problem that text-oriented console tools cannot directly work with MPS models is not solved generically. However, the most important one, checking and generating models, is supported. First, MPS models can be generated with an `ant` task. It transforms all models in a specified project, enabling subsequent compilation, test, and packaging of generated artifacts. Second, MPS supports a headless mode for executing type-system tests. These verify that error messages appear at locations in programs where, according to the type system rules, they should appear.

II.3: Requires tool support to export/import textual syntax By default, all *textual notations can be copy-pasted* to a text editor. The other way, from text to MPS, requires integrating a parser that creates the MPS tree from the textual source. MPS provides *hooks to integrate such parsers*. In mbeddr, we have developed more utilities for dealing with MPS code in the context of a text-based collaboration infrastructure. First, a node's ID can be represented as a text string, which can then be used by other developers to select the node in MPS. Second, a node can be copied as XML and then be transported via a text-based infrastructure. When the XML is pasted into MPS, the original node is reconstructed. This works independent of the notation.

5 Evaluation

We now evaluate the degree to which MPS' solutions of the drawbacks of ProjEs work in practice. The first two dimensions (Efficiently Entering (Textual) Code, Selecting and Modifying Code) are evaluated in Section 5.1 based on a survey; the questionnaire and anonymized results are available in [30]. Infrastructure Integration is discussed based on project experience in Section 5.2.

5.1 Editor Usability

Survey Setup Our survey addresses the following research question: *Does MPS solve known usability issues of projectional editors?* To answer it, we designed a questionnaire that assesses how developers work with MPS and how they perceive its usability. For each question, developers should estimate their opinion on a five-point Likert [31] scale, ranging from *strongly agree (1)* over *neutral (3)* to *strongly disagree (5)*. An example statements for developers to rate is: *I can work productively with MPS*. To help us understand the rating, we also asked users to elaborate on their rating in a text field. The survey questions are aligned with some of Nielsen's heuristic [32] to make sure the results are relevant for usability.

All participants are professional developers who are using or have used MPS for non-trivial tasks. We targeted professionals to obtain a controlled sample, excluding developers who have just experimented with MPS. We contacted each developer personally via e-mail. The contacted developers included users of mbeddr as well as other professional MPS users. Our contactees were allowed to forward the survey to other users. This led to one beginner in our sample, and we decided to not exclude the data so we can get an impression of the obstacles beginners face (to be explored further in future work). We piloted the questionnaire with one developer to rule out any misunderstandings in the questions: no adaptations were necessary, so we included the results from this developer in the analysis. To put the answers into context, we also assessed their general programming experience and how experienced they are with MPS and its underlying concepts (e.g., DSLs, AST, meta model, model transformation).

We used SurveyGizmo (`http://www.surveygizmo.com`) to present the questionnaire to developers. Completion took about 25 minutes, and developers were not compensated for their time. There are no deviations to report.

Participants We received responses from 21 developers, originating primarily from Europe (mostly Germany, the Netherlands, and Austria) and the US, plus one response from India. All have at least moderate experience with MPS. Eight of them have been using it between one and six months; only two just started, but three have used it for more than two years. The remaining seven developers report experience between half a year and two years. Most of the participants use MPS daily (13 developers), or at least multiple times a week (4); three less than once a week. 43% of developers estimate that they have written between 1,000 and 10,000 lines of code, only few (5%) less than 1,000, and many (29%) even more than 10,000 lines. Thus, our sample represents sufficient experience to establish an informed opinion about MPS.

Our participants have significant programming experience. Two thirds report more than ten years, with only one having less than two years. The experience as professional developer is also high (more than a third of participants report over ten years), but slightly lower on average, with five developers being beginners in professional development. Our most experienced participants were a managing director and a director of research and development with 24 years of experience. All but five participants have used a ParE-based IDE before, mostly Eclipse (62%) and Visual Studio (48%). The participants also have significant experience with model-driven development (MDD) and language engineering; this is not necessarily surprising, since MPS is a language engineering tool. For each of the nine MDD concepts (meta model, AST, grammar, DSL, textual DSL, graphical DSL, model transformation, M2T, and M2M), the majority reported being *very familiar*. The highest familiarity could be seen for meta model, AST, and textual DSL, while it was lower (but still on a high level) for graphical DSL and model transformation. Two thirds of all participants have used or designed a DSL before.

Usage Our participants report using MPS in a variety of domains – mostly automotive and embedded systems (these are the mbeddr users), but also the web, mobile, insurance, and enterprise resource planning. The majority of participants uses MPS as a programmer, while half of these also develop language extensions for mbeddr, indicating some more in-depth experience and language-design knowledge. One of them reported using it for his Master's thesis. Only three participants exclusively develop language extensions.

We now show the answers to how developers perceive various aspects of MPS, as assessed by the Likert-scale questions, shown in Fig. 2.

Efficiently Entering (Textual) Code (EE) Regarding Efficiency, most developers agree that they can write code as fast as with a ParE (median: 2; min/max: 1/5). Only one developer strongly disagrees, but explains that he is a proficient Emacs user ("Years of investment in Emacs are hard to beat."). One user also disagrees, but indicates that this is because he is a novice MPS user. A second developer who disagrees states that while code entering may not be that efficient, it is less error-prone, increasing overall efficiency. The remaining participants state that after getting used to the different style of entering code, there is no difference in efficiency to ParEs. We also asked about the general perception of Productivity with MPS. Most developers are positive in this respect (median: 2; min/max: 1/5). While 28% express a neutral opinion, 40% agree, and 28% even strongly agree. Only one participant expressed strong disagreement. This participant also faced intensive learning effort and stated that becoming familiar with the environment was difficult, mainly since all of MPS' concepts were completely new to him. In contrast, he strongly agrees that he can write code as fast as with a ParE, arguing that the code-completion facilities significantly contribute to the productivity. We conclude that after a learning phase, MPS lets developers work efficiently and productively.

Selecting and Modifying Code (SM) We asked developers what they think about producing correct programs with MPS (Correctness) and that they can produce only valid ASTs (CorrectAST). Most developers agree that these are supported well with MPS (median for both: 2). Many developers state that compared to a ParE, MPS does neither provide an advantage nor a disadvantage, because "the main type of errors are logical errors, which are not influenced by the IDE." Those who agree state that the error prevention in MPS is related to the fact that they can produce only valid ASTs. However, this enforcement of valid ASTs is also perceived as a drawback, because it reduces flexibility during programming: "Sometimes though, it would be nice to introduce classes, interfaces etc. by just using them, and then let the development environment generate the appropriate types if ordered so by the user via a quickfix".[5]

All participants agree that they benefit from the modular language support of MPS (median: 1, min/max: 1/2), confirming one of the key benefits of MPS. One developer states: "Language composition is the main strength of MPS." Regarding the support for different notations, the flexible notations provide a considerable benefit for developers (median: 2, min/max: 1/3), especially for integrating stakeholders from different domains ("My DSL users are business people, not IT people. Being able to use mathematical notations for Sum and Product expressions, fraction bars for division, tabular notations for test cases is crucial."). No problems were reported with the usability of these non-text editors.

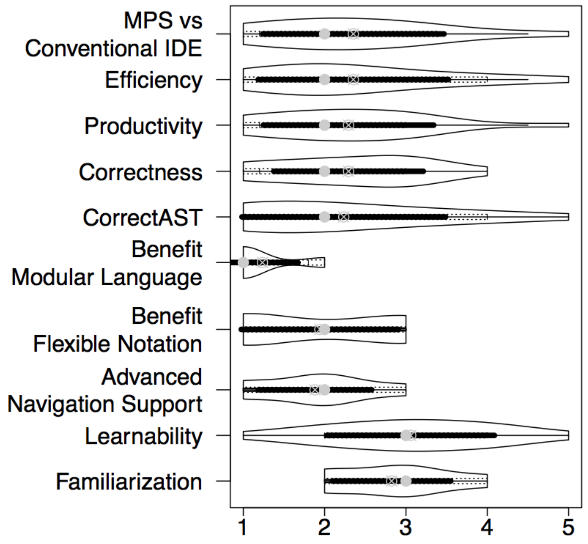

Fig. 2. Overview of Survey Answers.

Developers are often not satisfied with the commenting support of MPS, which is consistent with the shortcomings of the commenting facilities discussed earlier (Section 4.2). Developers complain about two main issues. The first one is the problems with (temporarily) commenting code ("You always have to use some workarounds, like cutting out program fragments ..."). The other one is the convenience of free text editing[6] inside documentation comments ("The editing of text is not straight forward ..."). We conclude that, except for supporting comments, MPS addresses the issue of selecting and modifying code quite well.

[5] mbeddr C provides such quick fixes (see Section 4, EE.2), but MPS' Java does not. This was a Java user.

[6] The plugin that supports unstructured free text editing for documentation and comments is a recent addition to MPS. Since the time of the survey, it has been improved significantly. It is now used to write the complete mbeddr user guide.

General Usability In addition to the three dimensions, we asked about the general usability of MPS. In general, developers like the advanced navigation support of MPS (median: 2; min/max: 1/3). Especially the direct navigation on the AST is a key advantage. This is especially true for the language engineers, as one user expresses: "Because of the direct navigation of the AST many features (refactoring, quick fixes, etc.) are easy to build." Participants expressed mixed feelings about learnability and familiarization. When asked whether MPS and its facilities are easy to learn and getting used to, more than half of our participants express a negative or neutral opinion, only few agree or strongly agree. Interestingly, the results become slightly more negative if we only consider participants who are both language users and language engineers (median: 4; min/max: 1/5). Thus, we conjecture that their perception was biased by the language development facilities in MPS, which require mastery of more advanced concepts than just *using* MPS languages to write programs. In fact, in the comment field, the respective participants reported only about issues related to language development. Just considering language users yields a more positive result (median: 3; min/max: 2/4, for both learnability and familiarization). One language engineer explained his positive attitude: "I have much experience and knowledge in language development, and given this background, MPS is rather logically structured. From this point of view, learning how to use MPS to build a new language [...] is not hard." Thus, with sufficient experience in language development, the learnability of MPS seems not to be a problem. Looking at more inexperienced programmers, we found that one problem of learning MPS is insufficient documentation, as stated by the same developer: "However, for certain specialist areas within MPS, there is a lack of good or enough documentation."

Since documentation is not a conceptual issue, we believe that learnability of MPS can be considerably improved with sufficient documentation. To address this, the MPS developers can build on the results of this survey.

Summary In general, the perception of MPS is positive. While the majority agrees that working with MPS can be productive, developers see some difficulty in learning (Learnability) and getting used to MPS (Familiarization). The overarching opinion regarding usability can be summarized in one sentence: MPS takes a while to get used to, but then its usability is comparable to ParEs. The stated advantages of ProjE, such as the flexible notations and modular languages, are also confirmed by our participants. However, there is also room for improvement: the hotspots expressed by the participants are in line with those problems identified in Section 3, for which MPS does not yet have satisfactory answers.

Threats to Validity To increase *external validity*, we ensured that all participants have significant prior experience with MDD. Thus, our survey results and conclusions about usability primarily apply to such developers. However, we had one beginner in our survey (participants were allowed to forward the survey to other MPS users). Thus, our results are slightly biased by this beginner, but at the same time give us valuable insights into the struggles that new MPS users face. We are currently planning a controlled experiment with students to further explore how beginners learn MPS. Regarding the results, we can carefully generalize beyond MPS based on the assumption that other ProjE can adopt MPS' usability-improving techniques, but further studies would improve the external validity. A threat to *internal validity* is that the results may be influenced by specific technical issues with MPS (or bugs), and are unrelated to the conceptual usability issues, as identified in Section 3. We mitigate this threat by targeting experienced developers, and cross-checking the experience with multiple questions in the survey questionnaire. To minimize biasing our participants, we asked them explicitly for the advantages and disadvangtes of MPS. By replicating our study, these threats can be reduced further. *construct validity*, i.e, to ensure that our survey measures usability correctly, we consulted the usability heuristics by Nielsen [32] before creating the questionnaire.

5.2 Infrastructure Integration

We now report on industrial experiences to evaluate the effectiveness of MPS infrastructure-integration support.

Version Control Since 06/2011, a team growing from five to eight people has been developing mbeddr based on MPS. Starting 07/2013, Siemens PLM has started developing a commercial tool on top of mbeddr, adding an additional four people to the team. Some of the mbeddr languages are also used in different domains, and two more developers are now working on the code base. This leads to a total of fourteen people. The work is spread over four git repositories[7]. In addition, two developers from BMW Car IT (plus two from mbeddr) worked on an SVN repository to develop an AUTOSAR extension for mbeddr.

In 2011 there were a few problems with merging; some changes just disappeared. This has since been fixed, and since 2012 no more problems have occurred with the VCS integration (git and SVN). Two aspects have to be kept on mind for it to work. First, diff and merge has to be done within MPS. Since all team members work with MPS anyway, this is not a problem. Second, if an update contains changes to languages as well programs that use these languages, users have to make sure to first merge and rebuild the languages. Otherwise MPS cannot correctly show the diff of programs written with these languages.

Continuous Integration mbeddr, as well as the projects built with it, use JetBrains Teamcity as an integration server[8]. It generates and compiles languages, runs tests, and packages the mbeddr system as MPS plugins. Even though Teamcity is also developed by JetBrains, there is no specific integration: it simply calls `ant` which in turn use MPS-provided `ant` tasks for building and testing.

Summary VCS integration and building on the server are the two most important concerns in terms of infrastructure integration. As discussed above, they are supported well. Used together with the mbeddr utilities for interoperability with textual environments discussed in Section 4.3, we conclude that infrastructure integration is addressed well enough to make MPS usable in practice. The mbeddr-specific extensions should be integrated directly into MPS, though.

[7] including the open source repo at `https://github.com/mbeddr/mbeddr.core`

[8] The CI server is at `https://build.mbeddr.com`; log in as `guest`.

6 Remaining Issues and Further Improvement

Automatically Deriving Actions The editor usability facilities have to be implemented manually for each language. While MPS provides DSLs to do this efficiently (and to a degree, generically), this is still tedious and error-prone. It is easy to forget some of the facilities for some language concepts, leading to an inconsistent user experience. One approach of addressing this problem is to describe textual-looking languages with a more grammar-like formalism from which many of the necessary editor facilities can be derived automatically. Both the MPS and mbeddr teams are currently experimenting with this approach.

Automatic Rebinding Consider a reference to a global variable `v`. If `v` is deleted, references to it break. Consider further that later, a new node named `v` is created, possibly a global variable or a function. The old references should now be bound to the new `v`. Currently, this is not supported; all reference sites have to be manually rebound by selecting the target from the code-completion menu. MPS 3.1 will support automatic rebinding of references based on target names stored in the (broken) references and existing scoping rules.

Legacy Import One use case of a ProjE is providing state-of-the-art IDE support and language extension and composition facilities for existing programming languages. To make this possible, the language must be reimplemented in MPS. The effort to do this is limited; it took the mbeddr team about five person months to implement C. However, in this scenario, interoperability with textual C code is necessary. Currently, a parser that creates MPS trees from text has to be implemented manually. If the aforementioned grammar-like formalisms were available, the necessary parser could potentially be automatically derived as well.

Command-line Support While MPS supports command-line integration for building and testing models, it is not possible to simply `grep` MPS models for text strings (beyond simple names). This is because programs are not stored in their concrete syntax notation. To address this problem, a textual representation of the program could be stored along with the AST-based persistence.

What you see is not what you type We are currently experimenting with two ideas for entering notations that are not on the keyboard. The first one simply shows the alias in a tooltip over the respective symbol. The second alternative uses a palette that contains buttons to enter those special notations.

Generic Commenting As confirmed by the survey, generic support for commenting (documentation as well as commenting out code) is necessary. Most likely this requires specific support by MPS' projectional editor. The MPS and mbeddr teams are currently discussing various approaches to the problem.

7 Conclusion

We have analyzed the usability of projectional editors, discussed mitigation techniques, and evaluated them by surveying professional developers. Our results show that the benefits of better language composition and notational flexibility are impaired by significant usability issues, but that the majority of those can be sufficiently mitigated with the facilities provided by MPS and discussed in this paper. In fact, the surveyed professional developers confirm the effectiveness of these mitigations in their every-day work, while the learning curve is high, requiring additional training. Further, our industrial experiences indicate MPS' usefulness for large-scale development projects. Thus, we believe that projectional editing can be efficient in projects that benefit from language composition and diverse syntax – outweighing the remaining usability issues. We believe this generalization is justified in the sense that MPS establishes a minimum viable set of techniques for improving editor usability that can be adopted by other ProjEs.

Our results can be used in various ways. The categorization of usability issues allows us to characterize ProjEs in general. The discussed mitigation techniques establish a minimal baseline for usability of ProjEs. Our empirical survey data indicates the cost (training and learning investment) to benefit (language composition, notational diversity, and potentially fewer errors) ratio, which can be used to assess the applicability of ProjEs in concrete projects.

Our future work is two-fold. First, we will investigate the remaining usability issues not currently addressed in MPS. Second, we aim at understanding adoption challenges, problem solving patterns, and efficiency with editing operations using a controlled experiment. It will comprise both beginning and professional developers (subset of survey participants), whose behavior when using a ProjE is compared to developers relying on a ParE. This experiment will complement our present work by providing an in-depth behavior analysis.

Acknowledgements We thank Alexander Shatalin, Vaclav Pech and Konstantin Solomatov for clarifying details about MPS and feedback to this paper.

References

1. Green, T.R.: Cognitive dimensions of notations. People and computers V (1989) 443–460
2. Simonyi, C.: The death of computer languages, the birth of intentional programming. In: NATO Science Committee Conference. (1995)
3. Czarnecki, K., Ulrich, E.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading, MA, USA (2000)
4. Medina-Mora, R., Feiler, P.H.: An Incremental Programming Environment. IEEE Trans. Software Eng. **7**(5) (1981)
5. Notkin, D.: The GANDALF project. Journal of Systems and Software **5**(2) (1985)
6. Porter, S.W.: Design of a syntax directed editor for psdl (prototype systems design language). Master's thesis, Naval Postgraduate School, Monterey, CA, USA (1988)
7. Reps, T.W., Teitelbaum, T.: The Synthesizer Generator. In: First ACM SIGSOFT-/SIGPLAN software engineering symposium on Practical software development environments, ACM (1984)
8. Voelter, M.: Language and IDE Development, Modularization and Composition with MPS. In: GTTSE 2011. LNCS. Springer (2011)
9. Simonyi, C., Christerson, M., Clifford, S.: Intentional Software. In: OOPSLA 2006, ACM (2006)
10. Christerson, M., Kolk, H.: Domain expert DSLs (2009) talk at QCon London 2009, available at `http://www.infoq.com/presentations/DSL-Magnus-Christerson-Henk-Kolk`.

11. Diekmann, L., Tratt, L.: Parsing composed grammars with language boxes. In: Workshop on Scalable Language Specifications. (2013)
12. Simi, M., Campagne, F.: Composable Languages for Bioinformatics: The NYoSh experiment. PeerJ PrePrints **1:e112v2** (2013)
13. Voelter, M., Ratiu, D., Tomassetti, F.: Requirements as first-class citizens. In: Proceedings of ACES-MB Workshop. (2013)
14. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education (2008)
15. Voelter, M., Ratiu, D., Kolb, B., Schaetz, B.: mbeddr: instantiating a language workbench in the embedded software domain. Automated Software Engineering **20**(3) (2013) 1–52
16. Voelter, M.: Preliminary experience of using mbeddr. In: 10th Dagstuhl Workshop on Model-based Development of Embedded Systems. (2014) 10
17. Coplien, J.O.: Multi-paradigm Design for C+. Addison-Wesley (1999)
18. Kats, L.C.L., Visser, E., Wachsmuth, G.: Pure and declarative syntax definition: paradise lost and regained. In: Proceedings of OOPSLA 2010, ACM (2010)
19. Parr, T.J., Quong, R.W.: ANTLR: A Predicated-LL(k) Parser Generator. Software: Practice and Experience **25**(7) (1995)
20. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF - reference manual. SIGPLAN **24**(11) (1989)
21. M. Bravenboer and E. Dolstra and E. Visser: Preventing injection attacks with syntax embeddings. In: GPCE 2007, Salzburg, Austria, ACM (2007)
22. Wyk, E.V., Bodin, D., Gao, J., Krishnan, L.: Silver: an Extensible Attribute Grammar System. ENTCS **203**(2) (2008)
23. Lämmel, R., Wachsmuth, G.: Transformation of sdf syntax definitions in the asf+ sdf meta-environment. Electronic Notes in Theoretical Computer Science **44**(2) (2001) 9–33
24. Anderson, R.: Two-dimensional mathematical notation. In Fu, K., ed.: Syntactic Pattern Recognition, Applications. Volume 14 of Communication and Cybernetics. Springer (1977)
25. Helm, R., Marriott, K., Odersky, M.: Building visual language parsers. In: Proc. ACM SIGCHI Conf. on Human Factors in Computing Systems. (1991) 105–112
26. Giammarresi, D., Restivo, A.: Two-dimensional languages. In Rozenberg, G., Salomaa, A., eds.: Handbook of Formal Languages. Springer (1997)
27. Pruša, D.: Two-dimensional context-free grammars. ITAT **2001** (2001) 27–40
28. van Rest, O., Wachsmuth, G., Steel, J.R.H., Süß, J.G., Visser, E.: Robust real-time synchronization between textual and graphical editors. In: Proceedings of ICMT 2013. Volume 7909 of LNCS., Springer (2013)
29. Voelter, M.: Integrating prose as first-class citizens with models and code. In: 7th International Workshop on Multi-Paradigm Modeling MPM 2013. (2013) 17
30. Online Appendix. `http://gsd.uwaterloo.ca/projectional-workbenches`
31. Likert, R.: A technique for the measurement of attitudes. Archives of psychology (1932)
32. Nielsen, J.: Usability Engineering. Morgan Kaufmann Publishers (1994)