Efficient Development of Consistent Projectional Editors using Grammar Cells

Markus Voelter
independent/itemis AG
voelter@acm.org

Tamás Szabó
Sascha Lisson
Bernd Kolb
itemis AG
{szabo|lisson|kolb}@itemis.de

Sebastian Erdweg
Delft University of Technology
s.t.erdweg@tudelft.nl

Thorsten Berger
Chalmers University of Gothenburg
thorsten.berger@chalmers.se

Abstract

The definition of a projectional editor does not just specify the notation of a language, but also how users interact with the notation. Because of that it is easy to end up with different interaction styles within one and between multiple languages. The resulting inconsistencies have proven to be a major usability problem. To address this problem, we introduce grammar cells, an approach for declaratively specifying textual notations and their interactions for projectional editors. In the paper we motivate the problem, give a formal definition of grammar cells, and define their mapping to low-level editor behaviors. Our evaluation based on project experience shows that grammar cells improve editing experience by providing a consistent and intuitive, "text editor-like" user experience for textual notations. At the same time they do not limit language composability and the use of non-textual notations, the primary benefits of projectional editors. We have implemented grammar cells for Jetbrains MPS, but they can also be used with other projectional editors.

1. Introduction

In projectional editors, a user's editing gestures directly change the abstract syntax tree (AST) of a program. Once changed, the projectional editor projects the AST to a suitable notation (or concrete syntax). This is in contrast to parser-based editors where users change the (textual) notation, and a parser builds the AST by recognizing structures in the sequence of characters.

Projectional editing has two major advantages: notational diversity and language composability. *Notational diversity* means that a wide range of notations can be used, including textual, tabular, diagrammatic and mathematical notations [27]. All notations are built on top of the same editor architecture, so these notations can be freely mixed (math symbols in tables, or text in diagrams) while retaining editor support for each notation. It is also possible to render the same program in multiple alternative notations. Notational diversity is crucial for DSLs targeting non-programmers such as insurance experts, systems engineers or biologists [31].

Language composition refers to using multiple languages in a single program without invasively modifying the definitions of the participating languages. Several forms of language composition exist [11], and projectional editors can support these composition techniques, as demonstrated in [26] for Jetbrains MPS[1], the currently most widely used projectional editor. Composition is simplified because the syntax of multiple languages used in a single program can never lead to parsing ambiguities. As a consequence, the abstract syntax, concrete notations, and IDE support of different languages always compose without problems (semantics can still be a challenge). An example of a set of language extensions based on a common base language is mbeddr [28], which includes over 30 modular extensions for C; the practical usefulness of this approach was shown in [30].

The main drawback of projectional editors is their questionable usability; in a previous paper [29] we identified issues in the areas of efficiently entering textual code, selecting and modifying code as well as infrastructure integration. In that paper we also described how MPS addresses these issues and we validated the degree to which MPS succeeds at addressing these problems with a survey among 20 experienced developers. It showed that usability is sufficient for encouraging developers to use MPS. In summary, for the participants of the survey, the benefits of language composition and mixed notations were not destroyed by the unusual editing gestures.

Yet,the survey in [29] and a follow-on student experiment [4] also identified issues that must be addressed in

Permission to make digital or hard copies of part or all 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 ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481. Copyright held by Owner/Author. Publication Rights Licensed to ACM.

Copyright © ACM [to be supplied]. . . $15.00

[1] http://jetbrains.com/mps

order to align editor behavior with the expectation of users, which is, that editing textual notations resembles, as much as possible, editing in traditional text editors. The first issue is that, in order to use a projectional editor efficiently, users need to have some understanding of the underlying AST structure of the edited program, because the tree structure governs editing behavior. The second issue is that editors of composed languages may exhibit inconsistent editor behaviors, confusing the user. For example, each language extension can define the behavior of pressing BACKSPACE on extension code. It is crucial that these behaviors are implemented consistently within and across languages. Otherwise, as the experiment in [4] shows, users may perform *over-deletions* when they inadvertently delete too much code or *invalid insertions* when they try typing code that is not recognized by the editor.

Contribution We introduce grammar cells, a new formalism for defining textual notations for projectional editors. Grammar cells provide a way of specifying projectional editors that preserves the advantages of projectional editing, but solves the code-modification problems identified in [29] and [4]. Achieving consistency within and between languages is simplified because of the declarative specification of editor behavior. We have implemented grammar cells in Jetbrains MPS, but they can also be used with other projectional editors.

Structure Sec. 2 illustrates the idiomatic problems in projectional editors and shows an example editor implementation with grammar cells. Sec. 3 provides a catalog of the grammar cells, and Sections 4 and 5 show their implementation with lower-level abstractions. Sec. 5 also evaluates critical performance aspects. Sec. 6 validates our approach by reviewing project experience with grammar cells. We discuss in Sec. 7 the implementation of grammar cells in MPS and in other tools. Sec. 8 presents related work, and Sec. 9 concludes.

2. Existing Editor Models by Example

A global variable declaration (GVD) in mbeddr C [28] adheres to the following syntax:

```
"exported"? "extern"? <type> <name> ("=" <init>)? ";"
```

The optional flags `exported` and `extern` mark the variable as being visible and defined outside the current module, respectively. A variable furthermore has a type, a name, and optionally an initializing expression. For example, this syntax supports the following declaration:

```
exported int8 x = 10 * y;
```

We now review how different existing editor architectures deal with this syntax in terms of editing behavior.

Text Editors In a classical text editor, users can enter the parts of a GVD in any order. They could start with the type, the name or any of the flags. The missing parts can be added later to form a syntactically correct declaration. This is the baseline expectation of users when using a textual notation in any editor. All established text editors and IDEs follow this approach.

Pure Projectional Editors In a pure projectional editor, users must first select the GVD concept from a code completion menu. Once selected, the editor projects the concept with "holes" at those places where the user can fill in the name, type and init expression:

```
<type> <no name> = <init>;
```

The optional flags are toggleable explicitly via a menu. Pure projectional editors are consistent in the sense that everything is done through code completion, menus and the "hole" metaphor; however, they do not support fluent editing like text editors. This does not satisfy user expectations as confirmed by our prior study [29] and others [19]. Examples of pure projectional editors include Prune [2] and the editor described by Clark [8].

Projectional Editor with Actions Actions are in-place transformations of the AST that are executed while the user edits the program, triggered by the user's editing activities. For example, an action could be programmed to create an empty `init` expression when the user types `=` on the right side of the `name` property in a GVD. This way, the actions improve the editing experience of projectional editors. However, they have two problems. First, the effort for implementing the necessary actions for all language concepts is a lot of work. Second, by implementing the actions in different ways, or by forgetting to implement some actions, it is very easy to create inconsistencies in the editing behavior within and between languages, thus confusing users. MPS is an example of a projectional editor with actions; as far as we know, the Intentional Domain Workbench [22] also provides some support for actions.

Problem Statement Our goal is to provide high-level constructs for the definition of flexible projectional editors that feature a user experience akin to text editors as much as possible. High-level constructs promote consistent editor definitions because they ease adoption and eliminate the variability in behavior induced by low-level tree actions. We aim at supporting at least the following user interactions for our GVD example:

1. Start with the `type`, then enter the `name`.
2. Start with `extern`, then enter `type` and `name`.
3. Start with `exported`, then enter `type` and `name`.
4. Start with `exported`, then continue with `extern`, `type` and `name`.
5. For an existing variable, type `exported`, `extern` or both on the left side of the type.
6. Optionally, type `=` and then enter the `init` expression on the right side of the `name`.

7. Delete the `exported` or `extern` flag by pressing Backspace on them.

Editing of expressions must also be improved. Consider an init expression `3 * 4 + x`. Users expect to be able to enter it linearly by typing "`3`", "`*`", "`4`", "`+`" and "`x`". This requires that the editor detects operator priority (or precedence) and constructs a tree corresponding to `(3 * 4) + x`. Now consider that the user wants to change the parenthesis-less expression to `3 * (4 + x)`. The user expects to be able to enter an opening parenthesis on the left of `4` and a closing parenthesis on the right of `x`. This interaction leads to a restructuring of the tree, called cross-tree editing, which we also aim to support.

More generally, we aim to provide high-level concepts for recurring editor patterns. Over the last 5 years, a team at itemis has spent roughly 30 person years developing MPS-based projectional editors for a wide range of real-world languages for embedded software, system specification, requirements engineering, safety and security analysis, insurance contract specification, medical software and public benefits calculations [28, 31]. Together with the results from our prior survey [29] and experiment [4], this allowed us to identify recurring editor patterns, for which the grammar cells described in this paper provide high-level abstractions.

3. The Grammar Cells Language

We introduce *grammar cells* as a high-level concept for the definition of flexible projectional editors for textual notations. In this section we discuss them from the perspective of the language engineer, i.e., the user of grammar cells. In the next two sections we switch the perspective to the implementor of grammar cells in a language workbench like MPS.

Fig. 1 (A) shows the editor definition for GVDs using grammar cells in MPS. An editor is defined for each language concept, and it specifies how instances of that concept are rendered in a program. Editor definitions are constructed from *cells*. A list cell (`[- -]`) contains sequences of other cells. Properties, such as `extern`, are embedded with the `{property}` syntax. Child cells, such as the `init` expression, use the `%child%` notation. Grammar cells contribute new kinds of editor cells for use in editor definitions. In addition to defining how a cell is rendered, each grammar cell also implies certain well-defined editor interactions that would otherwise have to be implemented manually.

In the rest of this section we explain each grammar cell in detail. We use boxes to show the structure of editor definitions where cells with a darker shade represent child cells of the lighter-shaded cell on their left. For example, in [C1|C2|C3], C2 and C3 are children of C1. Some actions are triggered when the user enters a particular character at a specific location in the program; we use the notation [l^C^r] to mark these positions. In the example, `l` represents the left side of the editor generated from the editor definition for the language concept `C`; `r` represents the right side.

The example in Fig. 1 (A) uses grammar cells `flag`, `wrap` and `optional`, resulting in an editor that supports all seven interaction scenarios for GVDs described in Sec. 2. The implementation of these grammar cells rely on lower level tree transformations actions, as explained in Sec. 4.

flag [flag|l^child(C.cld)] A `flag` cell represents a Boolean flag; the text is optionally shown in the program, depending on whether the flag is true or false. A flag cell surrounds the editor cell of a Boolean-typed child `cld` of a concept `C`. It enables setting the child by typing in a string, by default the name of the child link. It also allows unsetting the child by pressing DELETE or BACKSPACE on the editor cell marked with the position `l`.

In Fig. 1 (A) the editor cells of the `extern` and `exported` children are wrapped with flag cells which makes it possible to set these properties on a global variable by typing "extern" and "exported".

wrap [wrap|child(C.cld)] A `wrap` cell lets the user enter the child where the parent is expected, subsequently creating the parent. It wraps the editor cell of child `C.cld` and allows implicit instantiation of `C` by instantiating an instance of the concept in the `cld` link. The user disambiguates by selecting from the code completion menu if more than one concept can be created from the same child concept. Wrappers are also used for side transformations as we explain this.

In Fig. 1 (A) the editor cell of the `type: Type` child is nested in a wrap cell in order to allow the creation of a GVD by first typing in its `Type`.

optional [optional|l^const(t)|child(C.cld)] An `optional` cell lets the user enter an optional part of the syntax. It wraps a child editor cell; as long as `C.cld` is empty, the contents of an `optional` cell are not shown. Upon typing the string `t` at position `l`, the child is set to a non-null value, and the contents are shown.

In Fig. 1 (A) the initializer part of the global variable declaration becomes editable once the user types `=` on the right side of the `name` property in a GVD node.

The support for fluent, linear editing of expressions also relies on grammar cells. Fig. 1 (B) shows the generic editor definition for *all* binary expressions (`+`, `-`, `*`, `/`, `&&`, `||`, etc.). For the operands, we use `wrap` grammar cells to automatically generate the side transformations that let the user insert the operator tree when its symbol is entered to the left or right of an expression. We also use `constant` and `subsitute` cells:

constant constant A string computed from the underlying node or concept. For infix binary expressions, it displays the operator symbol (`+`, `*` or `||`).

substitute substitute | l^constant Surrounds a `constant` cell to support substituting the underlying concept with another one, picked from the code completion menu at position `l`. Proposals in the menu include those candidate concepts with similar structure (considering types and cardinality of children) as the current concept (cf. `structurallyMatches` in Sec. 4.2).

In our example, it supports changing the binary expression (e.g., changing `a * b * c` to `a + b * c`).

To handle priorities, associativity and cross-tree editing, the whole editor definition for binary expressions is nested in a `rule` grammar cell. When the content of a rule cell changes structurally, the editor linearizes the tree structure into a list of tokens and parses it into a new tree (we explain the details in Sec. 5).

rule ... Expresses that the contained cell structure should be processed using the built-in parser.

The rule of the binary expression in Fig. 1 (B) is defined as `%left% constant %right%` where `%left%` and `%right%` represent the children for the left and right hand side expressions and the `constant` represents the infix operator.

brackets brackets | l^const(o) | child(C.cld) | const(c)^r A bracket cell contains an opening bracket symbol `o`, a child cell `cld`, and a closing symbol `c`. Typing these symbols at positions `l` and `r`, respectively, inserts an instance of `C` (the concept that has the `brackets` editor) into the tree, restructuring it such that the subtree "between" the typed `o` and `c` is contained in the `cld` child.

Fig. 1 (D) shows the editor definition of the parenthesis expression. Supporting expression parenthesising is as simple as wrapping the inner expression in a brackets cell surrounded by the constants `(` and `)`.

splittable splittable | child(C.cld) A splittable cell wraps a child whose value can be split by typing specific characters in the middle of the literal. A splittable cell provides a tokenizer that returns the list of tokens after the value is split. The parser then uses the rules and the list of tokens to derive a new subtree.

Fig. 1 (C) uses the splittable cell for the number literal to allow splitting the value with operators, producing binary expressions (typing `+` in the middle of `44` returns `4 + 4`).

This 2-minute video demonstrates the resulting editor, for all editor cells discussed in this section, using the GVD as an example: https://youtu.be/QxXHtp90Fcs

4. Implementation of Grammar Cells

This section explains the implementation of the non-parser-based grammar cells. Our implementation is based on low-level actions available in MPS. They are typically used by programmers to manually implement well-behaving editors. However, as our experience over the last five years has shown, it is very hard to implement the actions for a set of languages completely and consistently – which has prompted the development of grammar cells. In other projectional editors where these low-level actions may not exist we still recommend implementing them as intermediate abstractions. We discuss the practicability of this in Sec. 7.

In the remainder of this section we formally define the low-level actions in Sec. 4.1 and then show the mapping from grammar cells to these actions in Sec. 4.2.

4.1 Low-Level Language

Fig. 2 shows the low-level MPS editor actions that we use in the implementation of grammar cells: substitutions, side transformations and delete actions. All of them are triggered by specific user editing gestures and then execute procedural Java code. The code is defined as a series of calls $\overline{fun}$ to helper functions. The definition of these functions is given in Sec. 4.2.

Substitutions A *substitution* for a concept C_1 lets the user create an instance c_2 of some other concept C_2 and then execute some code j. Typically, the code will create an instance c_1 of C_1 and set c_2 as a child of c_1. In the `GVD` example introduced earlier, a substitution is used to allow entering a `Type` when a `GVD` is expected (supporting scenario 1).

Side Transformations A *side transformation* on an instance c of concept C will be triggered when the user enters some string t at cursor position p. It then executes an action j. For example, a right transformation anchored on the `{name}` of a `GVD` is triggered by typing `=`; it sets the expression to something that is non-null (scenario 6). Similarly, entering a binary operator on the right or left of an expression inserts a binary expression. Note that in this case the priority and associativity of the various binary and unary operators have to be respected, and the necessary tree restructurings have to be implemented as part of j.

Delete Actions A *delete action* on an instance c of concept C will be triggered when the user presses BACKSPACE at cursor position p. It executes an action j that typically deletes c or sets a flag to false. Examples include actions that unset the `extern` and `exported` flags if BACKSPACE is pressed on them (scenario 7), as well as the removal of binary operators.

To illustrate these transformations in MPS, we conclude this section with two example transformations.

```
A  editor for concept GlobalVariableDeclaration
   [- flag{ exported } flag{ extern } wrap % type % { name } optional [- = % init % -] ; -]
B  editor for concept BinaryExpression
   rule: [- wrap % left % substitute constant wrap % right % -]
C  editor for concept NumberLiteral
   rule: [- wrap splittable{ value } -]
D  editor for concept ParensExpression
   rule: brackets[ ( % expression % ) ]
```

Figure 1. Editor definitions with grammar cells; (A) global variable declaration, (B) binary expression, (C) number literal and (D) parenthesis expression.

(substitution)	*subst*	::=	**subst**(| $c_2 : C_2 \mapsto \overline{fun}$)
(side transf.)	*side*	::=	**side**($c@p : C$ | $t \mapsto \overline{fun}$)
(deletion)	*delete*	::=	**delete**($c@p : C$ | $\mapsto \overline{fun}$)
(helper func.)	*fun*	::=	`reparse` | `reaplce` | `delete` | `nameOfLink` | `copyStructure` | `structuralMatches` | `new`

Figure 2. MPS-provided low-level actions onto which the grammar cells are mapped. The executable part of an action is defined with helper functions (Sec. 4.2).

```
1  substitute action wrapGVDwithType substitutes: GVD
2    wrapped: Type
3      (nodeToWrap, parentNode) -> node<GVD> {
4        node<GVD> var = new node<GVD>();
5        var.type = nodeToWrap;
6        return var; }
7
8  left transform actions leftTypeExported transforms: Type
9    condition: (model, sourceNode) -> boolean {
10     sourceNode.parent.isInstanceOf(GVD); }
11   action: add custom items  (output concept: Type)
12             matching text: "exported"
13             transform: (sourceNode, pattern)->void {
14               sourceNode.parent : GVD.exported = true; }
```

Figure 3. Two of the example actions; details are explained in the text.

Example Transformations Fig. 3, top, shows the wrapper rule that allows users to enter a `Type` when a `GVD` is expected (scenario 1). Line 1 specifies that the substituted node is of type `GVD`, and line 2 specifies that it is substituted by a `Type`. Lines 3-6 are invoked when the user enters a `Type`; the code procedurally creates a `GVD`, sets the previously entered `Type` as the `type` child of the `GVD` and then returns the newly created `GVD`.

Fig. 3, bottom, shows the transformation that sets the `exported` flag to true when a user types `''exported''` on the left side of a `Type` that is a child of a `GVD` (scenario 5). Line 11 specifies that the transformation applies to `Types`, and line 12 asserts that the `Type` is a child of a `GVD`. The action then matches the text "exported" (line 15) and then performs a transformation that sets the `GVD`'s `exported` flag to true (lines 16-17).

4.2 Translation of Grammar Cells

Fig. 4 shows the translation of grammar cells to low-level MPS actions. The rules specify the concept for which the editor is defined and additional constraints on the child elements and parent-child relationships. The `list` notation represents a collection of editor cells.

Helper Functions During the translation we use a set of helper functions:

`nameOfLink(`$C.cld$`)` Given a child `cld` of concept `C`, the function returns the label of C's edge that contains the child; in the example it would return "cld".

`new(`C`)` Creates a new instance of the concept C.

`delete(`c`)` Removes node c from the AST.

`replace(`$c_2 \leftarrow c_1$`)` replaces node c_1 in the AST with c_2.

`structuralMatches(`c_1`)` represents structural polymorphism [6] for MPS language concepts. It returns all language concepts C_2 which satisfy the following two conditions: (1) given a concrete instance $c_1 : C_1$, let Sup represent the concept which is the type of the containment edge that points to c_1 in the AST. Clearly, C_1 must be a subconcept of Sup and we enforce that C_2 is also a subconcept of Sup. (2) C_1 and C_2 must have identical node structure wrt. number, cardinality and type of children nodes.

`copyStructure(`$c_2 \leftarrow c_1$`)` $c_1 : C_1$ and $c_2 : C_2$ represent nodes where $c_2 \in structuralMatches(c_1)$. The function copies the children of c_1 and recreates the (similar) node structure under node c_2.

`reparse(c)` linearizes the children of node c into a list of tokens and rebuilds a subtree (if possible) using the grammar rule definitions of the concept C of c and the rules of its children (recursively).

Mapping We now explain the translation rules shown in Fig. 4 in more detail. The **flag** cell is defined for a concept `C`, and the wrapped child `cld` must have `Boolean` type. The flag cell results in a `side` transformation which is triggered when `nameOfLink(C.cld)` is typed at position `l` on a node `c: C`. The associated action sets the value of `c.cld` to true. A `delete` action is also generated, which sets `cld` to false when pressing BACKSPACE at position `l`.

An **optional** cell wraps a child `cld` of a concept `C`, where the child's type is another concept `T`. A right transformation triggered by typing the text `t` at the

flag	$C,\ C.cld\colon Boolean$ **in** [flag[l^child[C.cld]]]	$\Longrightarrow$	$\begin{cases} \textbf{side}(c@l : C \mid \texttt{nameOfLink}(C.cld) \mapsto c.cld = true) \\ \textbf{delete}(c@l : C \mid c.cld = false) \end{cases}$
optional	$C,\ C.cld\colon T$ **in** [optional[list[l^constant[t], child[C.cld]]]]	$\Longrightarrow$	$\begin{cases} \textbf{side}(c@l : C \mid t \mapsto c.cld = \texttt{new}\ T) \\ \textbf{delete}(c@l : C \mid \texttt{delete}(c.cld)) \end{cases}$
wrap	$C,\ C.cld\colon T$ **in** [wrap[child[C.cld]]]	$\Longrightarrow$	$\{\ \textbf{subst}(\ \mid t : T \mapsto c = \texttt{new}\ C,\ c.cld = t,\ \texttt{replace}(t \leftarrow c))$
subsitute	C_1 **in** [substitute[l^const]]	$\Longrightarrow$	$\begin{cases} \forall C_2 \in \texttt{structuralMatches}(C_1) : \\ \quad \textbf{subst}(\mathrm{c}_1@l\ :\ C_1\ \mid\ C_m.const\ \mapsto\ c_2\ =\ \texttt{new}\ C_2, \\ \quad\quad \texttt{copyStructure}(\mathrm{c}_2 \leftarrow c_1),\ \texttt{replace}(c_1 \leftarrow c_2)) \end{cases}$
brackets	$C,\ P,\ P.cld\colon D,\ C <: D$ **in** [brackets[l^constant[open], child[C.cld], constant[close]^r]]	$\Longrightarrow$	$\begin{cases} \textbf{side}(c@l : C \mid open \mapsto t : D = \texttt{reparse}(c),\ \texttt{replace}(c \leftarrow t)) \\ \textbf{side}(c@r : C \mid close \mapsto t : D = \texttt{reparse}(c),\ \texttt{replace}(c \leftarrow t)) \\ \textbf{delete}(c@l : C \mid t : D = \texttt{reparse}(c),\ \texttt{replace}(c \leftarrow t)) \\ \textbf{delete}(c@r : C \mid t : D = \texttt{reparse}(c),\ \texttt{replace}(c \leftarrow t)) \end{cases}$
Key for the notation: $C, C_1, C_2, D, P, T \in \mathbb{C}$ (*language concepts*) **in** [editor]		$\Longrightarrow$	$\textbf{action}(params \mid typed\ text\ \mapsto\ \texttt{executed code})$

Figure 4. Semantics of grammar cells defined via mapping to the low-level actions introduced in Fig. 2 and Sec. 4.1.

location of the optional cell is generated that instantiates a `T` and stores it in `cld`. The generated `delete` action lets the user delete `cld` at position `l` on `c`.

The context of the **wrap** cell is similar to that of optional. However, the wrap grammar cell results in a single `subst` action, which, when creating an instance `t` of `T`, it creates an instance `c` of `C` as well, sets the target of `cld` to `t` and replaces node `t` with `c`.

The **substitute** cell deals with replacing concepts. Whenever the code completion menu is queried at position `l` on an instance $c_1 : C_1$, it finds C_1's structurally matching concepts C_m. It then uses `subst` actions for all C_m to trigger a node structure replacement of c_1 upon selecting the element C_m from the completion menu by typing its operator associated constant $C_m.const$ (the operator symbol for binary expressions).

A **brackets** cell results in actions that allow inserting and deleting the left and right constants around the child `cld` of a concept `C`: a left transformation at position `l` with constant `open` and a right transformation at position `r` with constant `close`. In both cases, after the edits, the actions use the parser to build a new subtree and replace the existing node `c` with the new subtree rooted at `t`. The `delete` actions allow the user to delete the opening and closing constants at position `l` and `r` and take care of valid tree structures which represent the effect of the edits.

The **rule** cells are used for expression-like structures where trees must be restructured according to priority and associativity, and where cross-tree editing, such as those provided by the bracket cell, must be supported. The next section explains the details.

5. Linearization and Parsing

A major challenge in projectional editors is the linear editing of tree structures that are governed by priority and associativity rules (e.g., expressions) as well as modifying such trees in locations that do not lead to simple exchanges of single nodes (e.g., inserting or deleting parentheses in expressions). Our solution to this problem uses on-demand linearization of the respective (sub-)tree, plus subsequent reassembly through parsing [1, 14] that respects priority and associativity (see Fig. 5). This functionality is part of the **rule** cell.

Our parser is a non-directional depth-first search based parser (known as Unger's parsing method [24]). We chose this technique for three reasons. First, it is a simple algorithm with backtracking support due to its depth-first nature. Second, we must support extensibility of grammar rules, to enable language extensions without worrying about the parser. A parser which relies on a precomputed parsing table does not easily allow such extensibility. A depth-first search based parser, however, allows adding new grammar rules on-demand when a language extension is activated (as long as their is a way of dealing with ambiguity). Finally, the parser knows in advance the whole input because the linearization creates the full list of tokens from a subtree. This allows the use of a non-directional parser with essentially unlimited lookahead. Our optimizations (discussed later) rely heavily on accessing random elements of the input.

We have implemented our own parser because we do not parse text, so the mainstream parser generators are of no use. Our parser also exploits specifics of our tool (explained below), and no out-of-the-box parser knows about – and hence, exploits – these specifics.

5.1 Differences to Typical Parsers

Our use of parsing is very different from parsing text in a classical editor. We discuss the differences here.

As shown in Fig. 5, the linearized list of tree nodes is a temporary artifact: it is created from an existing tree through linearization. This means that we already start with tokens. There is no need to create tokens from

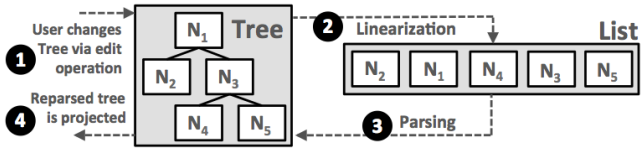

Figure 5. A parsing cycle comprises the user changing the tree (1), which is then linearized (2), reparsed according to structure, priority and associativity (3) and then projected back in the editor (4).

a sequence of characters (except in the **splittable** cell discussed above). We distinguish three kinds of tokens:

child A child token represents a node in the AST. It may have substructure which is parsed recursively.

constant A constant token either represents a string literal (such as a keyword or operator) or a primitive type-child of an AST node; primitive type children internally store their values as strings.

reference A reference token contains a pointer to an AST node that may not be part of the currently parsed subtree. Because it is a pointer, no recursive parsing happens.

Linearization proceeds along the AST structure, governed by the rule cells, and stops at nodes whose editor does not use a rule cell. Fig. 6 explains the details.

Note that parsing never creates new nodes (except for brackets), it only restructures existing nodes; new nodes are always created *directly* by a user's editing gestures (which is the distinguishing feature of projectional editors). The linearized version is created from this tree. This has two consequences. First, since a user's edit actions cannot create invalid trees (except wrt. to priority and associativity), the tree can always be reparsed. No error handling and error recovery must be supported.

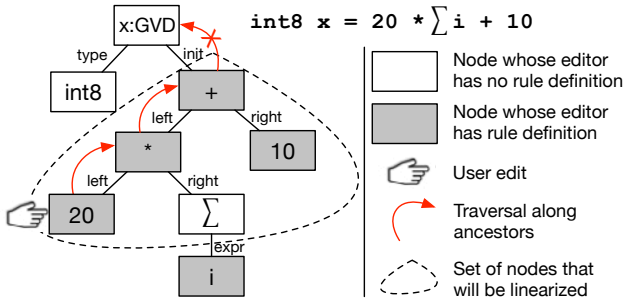

Figure 6. Linearization approach: after a change to a particular node (20 in the example), the linearizer traverses the tree up until the last node whose editor contains a rule; in our example, this is the + expression. Linearization then includes all descendants of this node whose editors include rules. Non-rule-editor nodes, such as ∑, are included in the linearized version as one token.

```
1  function pnode reparse(Node node, Concept concept)
2    return parse(linearize(node), concept)
3
4  function pnode parse(list<Tok> toks, Concept concept)
5    list<Rule> rules = get rules for all subconcepts of concept
6    rules = sort rules by priority
7
8    foreach rule in rules
9      set<SubRangeTuple> allSubRangeTups = split(toks, rule)
10     allSubRangeTups = filter allSubRangeTups by constants
11     allSubRangeTups = filter allSubRangeTups by ambiguity
12
13     if rule is left associative
14       allSubRangeTups = sort long to left allSubRangeTups
15     else
16       allSubRangeTups = sort long to right allSubRangeTups
17
18     label check:
19     foreach subRangeTuple in allSubRangeTups
20       root = new pnode(toks, rule.getSymbols())
21       foreach subRg, elem in subRangeTuple, rule.elements
22         if elem is child
23           if subRg.size == 1 && subRg.first is child
24             child = new pnode(subRange, elem)
25           else
26             child = parse(subRange, elem.concept)
27           if child != null
28               root.addChild(child)
29           else
30             continue check
31         else if (elem is constant)
32           if not(subRg.size == 1 &&
33               subRg.first is constant &&
34               subRg.first.value == elem.symbol)
35             continue check
36         else if (elem is reference)
37           if not(subRg.size == 1 &&
38              subRg.first is ref &&
39              isSubConcept(subRg.first.concept, elem.concept))
40            continue check
41         return new pnode(toks, rule.elements)
42   return null // parsing failed at this point
```

Figure 7. The parsing algorithm in pseudocode.

Second, even for nodes that are linearized into a string (such as operators), it is always known from which node they were created. This information is useful to resolve ambiguities. We explain this in detail below.

5.2 Parsing Algorithm

Fig. 7 shows the parsing algorithm in pseudocode. The `reparse` function represents the core of the cycle shown in Fig. 5 and is triggered after every user-initiated change of the AST (through code generated from the grammar cells). It linearizes the tree and then reparses it. We now explain the `parse` function in more detail, based on an example: the `a + b * c` expression.

The arguments of `parse` represent the current set of tokens that must be parsed, as well as the expected result concept. When we initially reparse `a + b * c`, the list of tokens would be `a, +, b, *, c` and the expected concept is `Expression`. Parsing is recursive; the `parse` function calls itself in line 26 (with different arguments). This shows the depth-first nature of the parser algorithm.

Collecting Candidate Rules In line 5 we collect all rules that are applicable to the expected concept, `Expression` in our case. Only those can be relevant

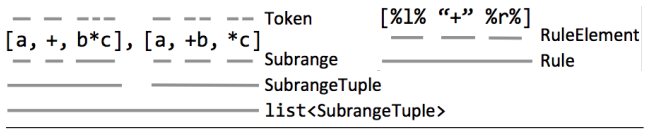

Figure 8. The nomenclature used in the text. *Rule*s are part of editor definitions and specify the syntax of language concepts. They consist of *RuleElement*s (children, constants and references). The linearized tree is split into a set of *SubRangeTuple*s. A SubRangeTuple consists of a list of *SubRange*s, each grouping *Token*s.

for parsing an instance of `concept`. To collect these rules, we enumerate all subconcepts of `Expression` that have a rule-based editor. In the example these are all direct and indirect subconcepts of `Expression` (several hundreds in mbeddr); among them `PlusExpression`, `MultiExpression` and `VarRefExpression`.

Priority In line 6 we sort them by priority. Recall that expressions with lower priority will end up further up in the tree, so in terms of the parser, they have to be recognized first. Line 6 thus sorts the rules by ascending priority. In the example, `PlusExpression` will come before `MultiExpression` in the list of sorted rules.

We then iterate over all the rules in line 8 and try to match them against the list of tokens. Whenever we find a match, we immediately create and return the subtree.

Splitting In line 9 we split the tokens (argument of `parse`) into a set of SubRangeTuples (see Fig. 8 for the nomenclature used here). One SubRangeTuple consists of as many SubRanges as the number of elements in a rule because each element of a rule must be matched with a SubRange of tokens while searching for a match. The algorithm tries all possible SubRangeTuples. Consider the `PlusExpression`'s rule, `%left% ‘‘+’’ %right%`: it has three elements, so we split the list of tokens into SubRangeTuples with three elements each: [`a,+,b*c`],[`a,+b,*c`],[`a,+b*,c`],[`a+,b,*c`], [`a+,b*,c`],[`a+b,*,c`]. As the number of possibilities grows quickly[2], we apply two optimizations that filter out possibilities which would not lead to a valid parse tree.

Line 10 rejects possibilities that are incompatible with the constant elements in the current rule. We start with this filter because it only requires string matching, which is cheap, and because in typical cases it leads to a significant reduction of possibilities. Of the six possibilities for `PlusExpression` we can immediately filter out all but one by observing that a constant rule element can only be matched with a SubRange of length 1 where the single token has the same value as the rule element (the `+` in this case). For example, the possibility [`a,+b,*c`] can never match, because at the position of the constant `+` element in the rule we find `+b` in the SubRangeTuple. Even though the possibility [`a+b,*,c`] has a SubRange of length 1 at the position of `+`, it is also rejected because the values are incompatible (`+` vs. `*`). This leaves [`a,+,b*c`] as the only possibility.

Ambiguity Line 11 deals with ambiguity by relying on the previously mentioned fact that the tokens are created from valid trees. Consider `a + b`: all tokens know that they originate from a `PlusExpression`. If another rule is available (for a different expression) that consumes the same tokens and has the same priority and associativity, parsing would be ambiguous (both reach line 11). However, *the user has already made the disambiguation* when he entered `PlusExpression`. Since the tokens have a reference to their originating concept, we can filter out the other alternatives due to mismatch with the tokens' context concepts. It is tempting to do ambiguity filtering before the loop in line 8 to avoid splitting altogether and improve performance. However, syntactically ambiguous expressions could be used as part of a subtree that is parsed in one go, thus we can only filter out incompatible splits specifically for a particular rule.

Associativity Lines 13 - 16 sort the SubRangeTuples according to associativity. For a left associative rule we prefer the possibilities with the longest leftmost SubRange, because these result in a subtree which resembles expressions that bind to the left. For right associativity, we sort by length of rightmost SubRange in descending order. In our example, we are already down to one possibility thus this sorting has no effect. However, in case of `a+b+c` as the original expression, the possibility [`a+b,+,c`] would be preferred over [`a,+,b+c`], thus respecting plus' left associativity.

Matching Line 19 iterates over all remaining SubRangeTuples, matching them against the current candidate `rule`. Line 21 is a parallel loop over the SubRanges and rule elements, matching each rule elements against the token types introduced earlier. There are three cases:

child If the rule element is a child (line 22) we branch further: for a SubRange with a single child token (e.g., `a`), we have found a match and we create a new parse node. Otherwise, if the SubRange has more than one element (e.g., `a*b`) it must be parsed and we call `parse` recursively. If either alternative creates a new node, we add it to the previously created root node. We have successfully constructed a (sub-)tree.

constant If the rule element is a constant (e.g., `+`) and the SubRange contains a single constant token with the same textual value we continue matching SubRanges against rule elements. Otherwise the

[2] In case of `n` tokens and `t` rule elements, the number of possible splits is $\binom{n-1}{t-1}$. This number grows quickly; it is the root cause of the performance behavior shown in Fig. 10.

current rule will not match, and we try the next possibility by jumping to the `check` label in line 34.

reference A reference is handled similarly to a constant, but instead of comparing the textual symbols, we enforce that the concept of the pointed-to node is a subconcept of the expected one.

We are now in line 41. If we have reached this point, we have matched a rule against a SubRangeTuple and we construct the corresponding parse result. We return, and unwind the recursive calls; if we have parsed a subtree, we end up returning to line 26, where we add the just constructed subtree to the containing `root`.

Finally, a `null` value is returned in line 43 if none of the rules matched the input tokens; no subtree replacement happens in this case. This happens, for example, if the input contains unbalanced parentheses. After the user fixes the problem by balancing the parentheses, a new parsing attempt is started that will then create a tree that respects the structure expressed by the parentheses.

5.3 Discussion

We introduce parsing into the projectional MPS editor because it provides better support for dealing with operator priority, associativity and cross-tree editing, while improving end-user experience and reducing the effort for language implementation. At the same time, we do not want to risk the two main benefits of projectional editing, non-textual notations and language composition. In this subsection we revisit these concerns.

Prioirity Line 6 sorts the rules in a way so that operators with higher priority are matched "further into" the parsing process, which makes them end up lower in the tree, thus encoding the higher priority. The language concepts underlying the tokens specify a numerical value for the priority.[3]

Associativity Lines 13 - 16 deal with associativity by ensuring that for left associative rules the SubRangeTuple with the longest leftmost SubRange is tried first; each language concept indicates its associativity.

Ambiguity The support for language composition and extensibility relies on the fact that there are never any problems with ambiguity. The reason for this is that, because program nodes are entered one at a time, "constructs with the same syntax" that would lead to ambiguities in parsers are disambiguated by the user at the time of entry. Despite our use of parsing, the tree is still modified by the user using the usual projectional in-place transformations; thus there is no problem with ambiguity *during entry*. However, there might be a problem during the reconstruction of the tree based on the token list. But as previously explained, each token knows its originating language concept, so we can filter SubRangeTuples that are incompatible with a given grammar rule.

This works well in practice, but it comes with one limitation. Consider two different multiplication concepts M_1 and M_2 that have the same grammar rule (same symbol, same concepts for left and right children) and a plus expression P which expects an M_2 as the left child. Suppose the user created the expression `a*b` as an instance of M_1 and wants to extend the expression to `a*b+c`. In the current implementation this is not possible because the context of the M_1 is not compatible with M_2 (recall that P expects an M_2). Without context-based filtering the parser could find a valid subtree that uses M_2 and P. In our implementation, the user must first "disambiguate" by changing M_1 to M_2 (easily achieved due to substitute cells) and then it is possible to type in the `+` symbol. We accept this limitation of context-based filtering because otherwise parsing may inadvertently change concepts, which we think is unacceptable. To illustrate this, consider a situation where the user composes languages which define concepts that can consume the same tokens but their priorities are different. If the user creates instances of the low-priority concepts and starts editing them, a parser without context restriction would start updating the user's program with instances of the high-priority concepts.

Non-Textual Notations The use of the parser does not prevent the use of non-textual notations. Recall that linearization only happens for nodes that use `rule` cells; the others are kept as single tokens in the linearized version of the tree. For example, in the case of a sum symbol in an expression $20 * \sum i + 10$, the whole sum (including limits and the body) is represented as a single token (cf. Fig. 6); the list of tokens would be `[20, *, <sum>, +, 10]`. No priority/associativity/parenthesis-based restructuring is possible into and out of such nodes; a rule-less concept acts as a barrier during linearization. However, this is also not expected; such expressions behave as if they were parenthesized: $20 * (\sum(i)) + 10$.

Parentheses Parentheses are typically encoded with a `ParensExpression` that projects the opening and closing symbols, and the parenthesized child expression between them. If the parser finds the corresponding opening and closing symbols, it creates such a `ParensExpression`. However, the intermediate stage, where a user has only entered the opening or closing symbol, cannot be represented as a valid tree. We solve this problem by storing unbalanced symbols in an annotation (nodes that are in the AST without the underlying concepts knowing of them). The side transformations that allow entering the symbols are generated from bracket cells (see Fig. 4). Only when two of them are entered in a

[3] A utility is available that reports the priorities of all language concepts used in a given set of languages.

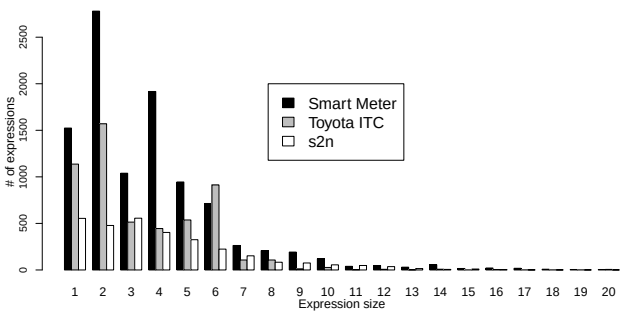

Figure 9. A histogram of the expression sizes. Over 97% of expressions have fewer than 10 sub-expressions. The maximum size is 50, but we cut the diagram because the y-value is so low that it cannot be recognized.

	N	Avg	Std. Dev.	Max
Smart Meter	10,000	3.82	3.21	40
Toyota ITC	5,399	3.36	2.22	20
Amazon s2n	3,033	4.09	3.06	50

Table 1. Expression sizes for three mbeddr C projects.

balanced way will the parser remove the annotations and create a `ParensExpression`. Deleting a symbol breaks up a `ParensExpression`, attaches annotations on corresponding nodes and restructures the tree as needed.

5.4 Performance

The performance of the parser is important because every tree change (i.e., in the worst case, every key press) requires execution of the parser (see Fig. 5). To find out whether the performance is acceptable, we have performed three steps. First, we have investigated the size of expressions in realistic code bases. Second, we have measured the parser performance for the expression sizes found. Third, we have experimented empirically with the editor, subjectively judging its responsiveness.

The Size of Expressions We have investigated three different mbeddr C systems regarding the size of expressions: the Smart Meter discussed in [30], the Toyota ITC static analysis benchmark[4] and a version of Amazon s2n.[5] Table 1 shows the results. We counted all expressions that had at least one child (a standalone variable reference was not counted), and we ignored expressions that contain multiple separately parsed expressions (such as array initializers or a fraction bar); the constituent expressions were counted as separate expressions, though. The biggest expression we found contained 50 subexpressions (a whole lot of flags were `or`'ed into a byte array), the overall average is around 4. Fig. 9 shows the distribution of expression sizes.

[4] https://github.com/regehr/itc-benchmarks

[5] https://github.com/awslabs/s2n

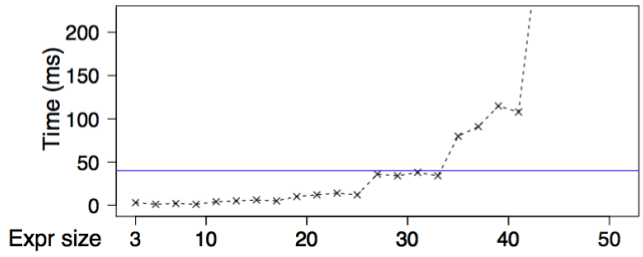

Figure 10. Reparsing time vs. expression size. Plot is cropped for space reasons; time for size 50 is 1067 ms. The blue horizontal line is at 40 ms.

Measurement We measured parser performance automatically using the following algorithm:

1. Start with an expression of size one (such as `10`)
2. Prepend one of `+, -, *, /` and another number (resulting, for example, in `20 * 10`)
3. Parse 10 times and calculate the average parse time
4. Repeat from 2. until the size of the expression is 50

The parse times are shown in Fig. 10: the relation of parse time vs. expression size is exponential. While this is generally bad news, it also shows that up to a size of 35, the parse time is below 40 ms (measured on a developer laptop, a 2.7 GHz Macbook Pro running OSX 10.11 and Java 1.8). The resulting delay during typing is not noticeable. This has also been empirically confirmed by interactively editing expressions of this size.

In addition to the size of the input, the parse time also depends on the number of language concepts (and hence, to-be-matched rules, see the loop in line 8 of Fig. 7). The measurements above have been performed in a program that includes all of mbeddr's C extensions; the numbers are the worst case.

Evaluation During our investigation of expression sizes we have only found 4 expressions with a size of more than 35 (including the one of size 50), which is 0.02 % of all expressions. Since these can always be refactored into smaller subexpressions (using local or global variables), and since expressions of this size are generally a bad idea regarding maintainability, we decided to ignore these expressions. We conclude that the performance of the parser is satisfactory and leave the investigation of further optimizations as future work.

6. Experience and Validation

The goal of grammar cells is to make it easier to build high-quality editors for textual notations where users can type code in a way that resembles text editors as much as possible. The necessary low-level code is generated in a consistent way in order to avoid surprising the user. Many of the problems reported in [4] (such as the invalid trailing insertions and over deletions mentioned in the introduction) can be traced back to inconsistencies in the manually implemented actions.

At the same time, the benefits of projectional editing – using non-textual notations mixed with text, as well as language modularity – must not be compromised. To validate the approach, we have not conducted another study or experiment like the ones in [29] and [4] *because grammar cells specifically fix issues found in this study*. However, we gathered significant experience from real-world projects:

Project Use All of mbeddr (C, 34 C extensions plus ca. 30 additional languages) have been ported to grammar cells. No problems with ambiguities or mixed notations have been found so far by us and our users. In addition, the feedback we received from our users regarding the consistency of the editors is positive. We have also used grammar cells in several other language development projects. In particular, we have taught them to new MPS language developers working for our customers. Using grammar cells, even these relatively novice users have been able to build good editors for MPS. Several of them called them a "game changer" and expressed that "they wouldn't use MPS without grammar cells".

Speed of Development As part of a new research project we have developed a new expression language from scratch. Relying on grammar cells, and in particular, the integrated parser, we have built the complete language and editor in an afternoon, including dealing with priority, associativity, parentheses support and splittable literals. While expression languages are not built very often (they are a prime candidate for reuse and extension), this is an impressive proof of the effectiveness of grammar cells: traditionally, building good expression languages has been a matter of several days and involved hundreds of lines of algorithmic, barely reusable code.

Limitations Grammar cell-based editors are still not *exactly* like text editors. Differences include: flags can only be added or removed completely (one cannot remove the `''rn''` of `extern`); typing `exported` when there are several different language concept whose editor can start with `exported` pops up the code completion to select the intended concept; and while sequences of flags can be entered in every order, they will always be projected in the order specified by the projection rules. However our experience indicates that these remaining differences are not perceived as problems.

7. Tool Integration

MPS MPS is bootstrapped, so the languages used by MPS for language definition are MPS languages themselves. This allows extending those languages with MPS' means for language extension. The grammar cells are an extension of MPS' editor definition language. Similar to the extensions for mathematical notations, tables or diagrams [27], the grammar cells language defines new cell types that can be used in MPS editor definitions. From these, we generate regular MPS editor cells plus the necessary low-level actions.

Other Tools We have only implemented grammar cells in MPS. However, we believe the approach could also be used in other projectional editors, such as those mentioned in [12] or [8]. In projectional editors supporting the low-level transformations introduced in Sec. 4.1, the grammar cells can be added directly, assuming that the syntax definition language is extensible to include the necessary markup. If these low-level actions do not exist, our discussion in Sec. 4.1 should provide enough detail to implement them. For this to work, the only precondition is the editor's ability to hook into keyboard events (such as pressing `+` on the right side of an expression). However, since a projectional editor relies on these keyboard events to modify the tree in the first place, this requirement is easily met.

8. Related Work

Language Workbenches All contemporary projectional editors are part of language workbenches, i.e., tools that allow users to define, compose and use their own languages [13]. As shown in [12], 4 out of the 10 tools that took part in the 2013 Language Workbench Challenge are projectional editors. At the time of the challenge, both Onion and Más [5] were in very early stages of development and did not provide and support for grammar cell-style specification of usable editors. The tools have since been discontinued. The Whole Workbench [23] emphasizes structured notations (trees, tables) and does not emphasize usable textual editors.

Projection and Usability An early example of a projectional editor is the Incremental Programming Environment (IPE) [16]. 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 did not address editor usability; to enter `2+3`, users first have to enter the `+` and then fill in the two arguments. Another early example is GANDALF [17]; the report in [19] states that the authors experienced similar 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 Intentional Programming project [9, 21] 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 [7, 22]. Since it is a commercial system, we cannot evaluate its usability, and whether facilities similar to grammar cells are available. What is known from the above-mentioned publications suggests that this is not the case.

Clark describes a proof-of-concept projectional editor [8] based exclusively on tree transformations. No emphasis has been put on usability. However, since the low-level tree change events are available, grammar cells could definitely be integrated.

Scratch [15] is an environment for learning programming. It uses a projectional editor, but does not focus on textual editing; it relies mostly on nested blocks/boxes. Prune [2] is a projectional code editor developed at Facebook. The goal is explicitly to *not* feel like a text editor; the hypothesis is that tree-oriented editing operations are more efficient than those known from text editors. While this is an interesting hypothesis, our considerable experience with using projectional editing in real projects has convinced us that this approach is not feasible; hence the work described in this paper.

Projection and Parsing The Synthesizer Generator [20] is a projectional editor which, at the fine-grained expression level, uses textual input and (regular, textual) parsing. While this improves usability, it destroys many of the advantages of projectional editing in the first place, because language composition and the use of non-textual notations *at the expression level* is limited.

Eco [10] relies on language boxes, explicitly delineated boundaries between different languages used in a single program (e.g., the user could define a 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 projectional editor. 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 without the need to constantly switch boxes manually, as well as integrated symbol tables, may not be a trivial problem. More generally, Eco has been developed with a background in parsing, trying to get some of the advantages of a projectional editor through language boxes. Our work starts out from projectional editing, trying to get to a more parser-like editor experience. A systematic and in-depth comparison of the trade-offs between the two approaches would be an interesting exercise.

To the best of our knowledge the use of Unger's parsing method in a projectional workbench is unique; we did not find other related solutions.

Language Compositions with Parsers Parser-based systems mitigate ambiguities by some form of disambiguation logic. This is fundamentally different from projectional editors which rely on the user's explicit choice for disambiguation. Our embedded parser relies on the choice of the user when reparsing a program fragment; no additional disambiguation logic is needed.

ANTLR [18], a parser generator for LL(k) grammars, allows a grammar to extend one other grammar, composition of several grammars is not supported. Resulting ambiguities must be handled by (invasively) refactoring the grammar. In some cases, syntactic predicates are sufficient. Blender [3] relies on a GLR parser and supports full context-free languages. It supports modular language composition and embedding. Ambiguities are handled through the underlying lexer's longest matching analysis and on the ordering of production rules. However, this becomes impractical when multiple separately implemented grammars are composed.

The Syntax Definition Formalism (SDF) [25] relies on a scannerless GLR parser and supports full context-free languages. SDF provides declarative constructs to deal with ambiguities: specifying priority and associativity of production rules, preferences, rejections and restrictions. SDF organizes productions into modules, supporting modular grammar composition. If disambiguation logic is required, it can be defined in a separate module, no invasive change to composed modules is required.

All of the previous approaches become impractical if multiple languages are composed, because disambiguation may be needed between all of them. This is reinforced by our observation that parser-based techniques have not been used to build a system of dozens of composable language extensions like mbeddr [28].

9. Conclusion

In this paper we have described grammar cells, a formalism for defining usable and consistent editor behaviors for textual notations in projectional editors. In particular, grammar cells generate the low-level behavioral code that let users comfortably enter and delete program nodes. Some cells rely on parsing to handle priority, associativity and cross-tree editing for expressions. We have successfully implemented and evaluated grammar cells based on MPS; however, the approach could also be used with other projectional editors.

Using grammar cells, projectional editors better meet users' expectations of behaving like text editors for textual notations. The implementation effort for such usable editors is much lower than using traditional approaches. At the same time, the core benefits of projectional editors, language modularity and the use of non-textual notations, are not compromised.

Ultimately, this will help with the adoption of projectional editors in practice, bringing a wide variety of languages to diverse application domains.

References

[1] A. V. Aho and J. D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., 1972.

[2] K. Beck and T. Hirai. Prune: A Code Editor that is Not a Text Editor. https://www.facebook.com/notes/kent-beck/prune-a-code-editor-that-is-not-a-text-editor/1012061842160013/, 2015.

[3] A. Begel and S. Graham. Language analysis and tools for input stream ambiguities. In *Fourth Workshop on Language Descriptions, Tools and Applications (LDTA'04), Electronic Notes in Theoretical Computer Science, Barcelona, Spain*, 2004.

[4] T. Berger, M. Voelter, H. P. Jensen, T. Dangprasert, and J. Siegmund. Efficiency of Projec-tional Editing: A Survey, an Experiment, and Lessons Learned. In *Accepted for Publication at the International Symposium on the Foundations of Software Engineering, FSE 2016*, 2016.

[5] M. Boersma. Más Workbench. http://www.mas-wb.com/, 2013.

[6] L. Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 70–79. ACM, 1988.

[7] M. Christerson and H. Kolk. Domain expert DSLs, 2009. talk at QCon London 2009, available at http://www.infoq.com/presentations/DSL-Magnus-Christerson-Henk-Kolk.

[8] T. Clark. A general architecture for heterogeneous language engineering and projectional editor support. *ArXiv 1506.03398*, 2015.

[9] K. Czarnecki and E. Ulrich. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, MA, USA, 2000. ISBN 0201309777.

[10] L. Diekmann and L. Tratt. Eco: A language composition editor. In *Software Language Engineering*, pages 82–101. Springer, 2014.

[11] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language Composition Untangled. In *Proceedings of LDTA*, 2012. to appear.

[12] 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 *Software Language Engineering*, pages 197–217. Springer, 2013.

[13] M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005.

[14] D. Grune and C. J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood, Upper Saddle River, NJ, USA, 1990. ISBN 0-13-651431-6.

[15] C. M. Lewis. How programming environment shapes perception, learning and goals: Logo vs. scratch. In *41st ACM Technical Symposium on Computer Science Education*, 2010.

[16] R. Medina-Mora and P. H. Feiler. An Incremental Programming Environment. *IEEE Trans. Software Eng.*, 7(5), 1981.

[17] D. Notkin. The GANDALF project. *Journal of Systems and Software*, 5(2), 1985. doi: http://dx.doi.org/10.1016/0164-1212(85)90011-1.

[18] T. Parr, S. Harwell, and K. Fisher. Adaptive ll (*) parsing: the power of dynamic analysis. *ACM SIGPLAN Notices*, 49(10):579–598, 2014.

[19] S. W. Porter. Design of a syntax directed editor for psdl (prototype systems design language). Master's thesis, Naval Postgraduate School, Monterey, CA, USA, 1988.

[20] T. W. Reps and T. Teitelbaum. The Synthesizer Generator. In *First ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*. ACM, 1984.

[21] C. Simonyi. The death of computer languages, the birth of intentional programming. In *NATO Science Committee Conference*, 1995.

[22] C. Simonyi, M. Christerson, and S. Clifford. Intentional Software. In *OOPSLA 2006*. ACM, 2006. ISBN 1-59593-348-4.

[23] R. Solmi. Whole Platform. http://whole.sourceforge.net, 2013.

[24] S. H. Unger. A global parser for context-free phrase structure grammars. *Commun. ACM*, 11(4):240–247, Apr. 1968. ISSN 0001-0782.

[25] E. Visser et al. *Syntax definition for language prototyping*. Eelco Visser, 1997.

[26] M. Voelter. Language and ide development, modularization and composition with MPS. In *GTTSE 2011*, LNCS. Springer, 2011.

[27] M. Voelter and S. Lisson. Supporting Diverse Notations in MPS' Projectional Editor. *GEMOC Workshop*.

[28] M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20 (3):1–52, 2013.

[29] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. Towards User-Friendly Projectional Editors. In *7th International Conference on Software Language Engineering (SLE)*, 2014.

[30] M. Voelter, A. v. Deursen, B. Kolb, and S. Eberle. Using c language extensions for developing embedded software: A case study. In *Proceedings of OOPSLA 2015*, pages 655–674. ACM, 2015.

[31] M. Voelter, J. Warmer, and B. Kolb. Projecting a modular future. *Software, IEEE*, 32(5):46–52, 2015.