Efficient Development of Consistent Projectional Editors using Grammar Cells

Markus Völter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, Thorsten Berger
Why Projectional Editors
[Projectional Editing]

Parsing

- Hand
- Eye
- Concrete Syntax
- Abstract Syntax Tree

Projectional Editing

- Hand
- Eye
- Concrete Syntax
- Abstract Syntax Tree
### Projectional Editing

**Syntactic Flexibility**

#### Regular Code/Text

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Mathematical

\[ \sum \]

#### Tables

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Graphical
[Projectional Editing]
Syntactic Flexibility

Regular Code/Text

```c
// A documentation comment with references
void aSummingFunction(int8[] data, int8 dataLen) {
    int16 sum;
    for (int8 i = 0; i < dataLen; i++) {
        sum += data[i];
    }
    for
} aSummingFunction (function)
```

Mathematical

```c
double midnight2(int32 a, int32 b, int32 c) {
    return \frac{-b + \sqrt{b^2 - \sum_{i=1}^{4} a * c}}{2 * a};
} midnight2 (function)
```

Tables

```c
int16 decide(int8 spd, int8 alt) {
    return spd > 0 spd > 100 otherwise 0;
    alt < 0 | 1 | 1
    alt == 0 | 10 | 20
    alt > 0 | 30 | 40
    alt > 100 | 50 | 60
} decide (function)
```
[Projectional Editing]
Language Composition

Separate Files
- Type System
- Transformation
- Constraints

In One File
- Type System
- Transformation
- Constraints
- Syntax
- IDE
[Projectional Editing]
Language Composition

Embedding

\[ L_{Host} + L_{Adapt} + L_{Emb} = \]

Extension

\[ L_{Base} + L_{Ext} = \]

Extension Composition

\[ L_{Base} + L_{Ext1} + L_{Ext2} = \]
The Usability Issue
Study Results on Editor Usability

People prefer MPS over conventional IDEs
MPS more is more efficient than normal IDEs
MPS more is more productive than normal IDEs
MPS makes it easier to create correct programs
MPS enforces a structurally correct AST
People benefit from language modularity
People benefit from the flexible notations

The experience with learning MPS is mixed.
It takes some time to get used to MPS

Strongly agree ... Neutral ... Strongly disagree
Early Days

The tree dominated the editing experience. Enter new nodes based on tree structure. Select and modify based on tree structure. Modify through menu-based user interactions.

No user acceptance because too slow, and not like text editing for textual notations.
1980  1990  2000  2010

Resurgence

Textual Notations can be edited „linearly“. Based on little tree-transformations triggered by editing actions. Those actions had to be built manually. Effort for good editors is high. Inter|ra}-Language Consistency is a problem.

User acceptance was mostly there, but few good editors ever built.
Textual Notations can be edited „linearly“. Based on little tree-transformations triggered by editing actions. Actions automatically derived from higher-level semantically rich editor descriptions.

Effort for building good editors has gone to almost zero. Editors are consistent.
Grammar Cells

Enter new nodes based on tree structure.

Enter nodes mostly linearly/textually.

Modify through menu-based user interactions

Modify mostly through typing, deleting, etc.

Effort for good editors is high.

Reduced through abstraction & code generation.

Int{er|ra}-Language Consistency is a problem.

Consistency is there b/c of idiomatics.

Select and modify based on tree structure.

(This issue is still there, unchanged.)
Grammar Cells Demo
https://www.youtube.com/watch?v=QxXHtp90Fcs
How Grammar Cells Work
More semantics in the editor definitions

<table>
<thead>
<tr>
<th>A</th>
<th>editor for concept</th>
<th>GlobalVariableDeclaration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>flag{ exported }</td>
<td>flag{ extern }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>editor for concept</th>
<th>BinaryExpression</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule:</td>
<td>[- wrap</td>
<td>% left</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>editor for concept</th>
<th>NumberLiteral</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule:</td>
<td>[- wrap</td>
<td>splittable{ value }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>editor for concept</th>
<th>ParenExpression</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule:</td>
<td>brackets[ ( % expression % ) ]</td>
<td></td>
</tr>
</tbody>
</table>

Based on problems identified in user studies and accumulated experience from dozens of developers and languages.
Declarative Descriptions for the most typical editor actions

flag  
\[ C, \ C.cl\!\!d: \text{Boolean} \ \text{in} \ [\text{flag][}^{\text{child[C.cl\!\!d]}]}] \]

optional  
\[ C, \ C.cl\!\!d: \ T \ \text{in} \ [\text{optional}[\text{list}[I^{\text{constant[t]}}, \ \text{child[C.cl\!\!d]}]]] \]

wrap  
\[ C, \ C.cl\!\!d: \ T \ \text{in} \ [\text{wrap[child[C.cl\!\!d]]]}] \]

substitute  
\[ C_1 \ \text{in} \ [\text{substitute}[I^{\text{const}}]] \]

brackets  
\[ C, P, P.cl\!\!d: D, C <: D \ \text{in} \ [\text{brackets}[I^{\text{constant[open]}}, \ \text{child[C.cl\!\!d]}, \ \text{constant[close]^r}]] \]

Key for the notation:
\[ C, C_1, C_2, D, P, T \in C \ (language \ concepts) \ \text{in} \ [\text{editor}] \ \rightarrow \ \text{action}(params \mid typed \text{ text} \rightarrow \text{executed code}) \]
Declarative Descriptions
for the most typical editor actions
translated to the available action primitives in PE

flag
\( C, C.cld: Boolean \) in \([\text{flag}[^{\text{^child}}[C.cld]]]]\) \( \implies \) \( \begin{cases} \text{side}(c@l:C | \text{nameOfLink}(C.cld) \mapsto c.cld = \text{true}) \\ \text{delete}(c@l:C | c.cld = \text{false}) \end{cases} \)

optional
\( C, C.cld: T \)
in \([\text{optional}[\text{list}[^{\text{^constant}}[t], \text{child}[C.cld]]]]\)
\( \implies \) \( \begin{cases} \text{side}(c@l:C | t \mapsto c.cld = \text{new } T) \\ \text{delete}(c@l:C | \text{delete}(c.cld)) \end{cases} \)

wrap
\( C, C.cld: T \) in \([\text{wrap}[\text{child}[C.cld]]]]\)
\( \implies \) \( \begin{cases} \text{subst}( | t: T \mapsto c = \text{new } C, c.cld = t, \text{replace}(t \leftarrow c)) \\ \forall C_2 \in \text{structuralMatches}(C_1): \begin{cases} \text{subst}(c_1@l:C_1 | C_m.const \mapsto c_2 = \text{new } C_2, \\ \text{copyStructure}(c_2 \leftarrow c_1), \text{replace}(c_1 \leftarrow c_2)) \end{cases} \end{cases} \)

substitute
\( C_1 \) in \([\text{substitute}[^{\text{const}}]]\)
\( \implies \) \( \begin{cases} \text{subst}(c_1@l:C_1 | C_m.const \mapsto c_2 = \text{new } C_2, \\ \text{copyStructure}(c_2 \leftarrow c_1), \text{replace}(c_1 \leftarrow c_2)) \end{cases} \)

brackets
\( C, P, P.cld: D, C <:: D \)
in \([\text{brackets}[^{\text{constant}}[\text{open}], \\ \text{child}[C.cld], \text{constant}[\text{close}[^{\text{r}}]]]]\)
\( \implies \) \( \begin{cases} \text{side}(c@l:C | \text{open} \mapsto t:D = \text{reparse}(c), \text{replace}(c \leftarrow t)) \\ \text{side}(c@r:C | \text{close} \mapsto t:D = \text{reparse}(c), \text{replace}(c \leftarrow t)) \\ \text{delete}(c@l:C | t:D = \text{reparse}(c), \text{replace}(c \leftarrow t)) \\ \text{delete}(c@r:C | t:D = \text{reparse}(c), \text{replace}(c \leftarrow t)) \end{cases} \)

Key for the notation:
\( C, C_1, C_2, D, P, T \in C \) (language concepts) in \([ \text{editor} ]\)
\( \implies \) action(params | typed text \mapsto \text{executed code})
Integrated Parsing for expressions to deal with precedence, associativity and cross-tree editing.
Integrated Parsing for expressions

complex, non-text tokens remain intact. Notation mixing still possible.

\[
\text{int8 } x = 20 \times \sum i + 10
\]
Wrap Up
PEs have many advantages. Mixing Notations, Language Composition.

Editing Experience was a Challenge. Textual Notations were not editable as in text editors.

Editor behavior must be consistent within one and across several (composed) languages.

Grammar Cells support „nice“ editors with very limited editor development effort.

Editor End-User Language Dev } feedback is very positive „this changes the game for Projectional Editors“
PEs have many advantages.
Mixing Notations, Language Composition.

Feedback is very positive
```
"this changes the game for Projectional Editors"
```

Grammar Cells make exploiting these benefits a real option!

Editor End-User
Language Dev

} feedback is very positive