Projecting a Modular Future

Markus Voelter[1], Jos Warmer[2], and Bernd Kolb[3]

[1] independent/itemis, voelter@acm.org
[2] independent, jos.warmer@openmodeling.nl
[3] itemis, bernd.kolb@itemis.de

Abstract. We describe two innovations in programming languages: modularity and projectional editing. Language modularity refers to the ability to combine independently developed languages without changing their respective definitions. A language is not anymore a fixed quantity, instead it can be extended with domain-specific constructs as needed. Projectional editing refers to a technique of building editors and IDEs that avoid the need for parsers. They support a wide range of tightly integrated notations including textual, symbolic, tabular and graphical. In addition, by avoiding parsers, the well-known limitations of grammar composition are avoided as well. The article illustrates the consequences of these two innovations for the design of (programming) languages with three examples. First, we discuss a set of modular extensions of C for embedded programming that enables efficient code generation and formal analysis. Second, we discuss a language for requirements engineering that flexibly combines structured and unstructured (prose) data. Third, we illustrate a language for defining insurance rules that makes use of mathematical notations. All examples rely on the open source JetBrains MPS language workbench.

Keywords: Language Engineering, Language Workbenches, Projectional Editing, Domain-Specific Languages, Programming Languages

1 Introduction

In modular languages, the abstractions provided by the language are *not* fixed. Users can pick language extensions from a library and include them into their programs without changing the host language definition or the IDE. Multiple independently developed extensions can be used together, and new extensions can be developed and used at any time. A classification of language composition approaches is discussed in [1].

A promising approach to developing modular languages is to use *language workbenches*. LWBs are environments that support the efficient implementation of languages and associated tools such as type checkers, compilers, interpreters and IDEs. The term LWB has been introduced by Martin Fowler [2] in 2004, even though such tools can already be found in the 80s and 90s (examples include the Synthesizer Generator [3] or the Meta Environment [4]). Contemporary examples include Rascal [5], Spoofax [6] or MPS, discussed below. An overview and comparison of today's LWBs can be found in [7].

JetBrains MPS[4] is a open source LWB licensed under Apache 2.0. It has been developed by JetBrains over the last 10 years, based on the initiative of Sergey Dmitriev. MPS is used in a number of projects in computational biology [8], web applications[5], embedded software development, requirements engineering and insurance DSLs (the latter three are discussed later in this article). MPS' most distinguishing feature is its projectional editor.

Languages typically use either textual and graphical notations; each style comes with different user experiences and use cases, and their editors also use different architectures. In textual languages, users interact with the concrete syntax, entering characters into a text buffer. A parser then matches the sequence of characters to the grammar that defines the syntax of the language, and constructs an abstract syntax tree (AST) of the program. The AST contains much more structure than the flat textual notation. Even though modern IDEs construct the AST in realtime as the user edits the program (maintaining an always-up-to-date AST), users interact with the textual source. Graphical editors are different. If a user, for example, drops a UML class from the palette onto the diagram, the underlying tool *directly* modifies the AST (aka the *model* in graphical editors). A rendering or projection engine then creates a visual representation of the AST. This approach can be generalized beyond graphical notations: the result is called a *projectional* editor (PE). Relative to textual-looking notations, it is important to understand that *every single text string is immediately recognized as it is entered*, so there is never any "extraction" of the AST from the concrete syntax by a parser.

This has a number of advantages. Since there is no need to extract the structure of the program from a flat (textual) source, a wide variety of notations can be used. MPS, for example, supports textual notations, symbols (such as fraction bars or $\sum$), tables as well as graphical diagrams; these notations can also be mixed. As we show below, this notational freedom enables languages that are much closer to the established notations of many application domains.

The implementation of modular languages is also simplified. Since no grammars are used, the limitations of composability known from grammars (and described wonderfully in [9]) do not apply. Of course, composition of languages still requires the alignment of the semantics (which can be a challenge), but from a purely syntactical perspective, there is no limit to composability.

Traditionally, PEs also have drawbacks, which is why they have not seen much adoption despite their advantages. What distinguishes MPS from earlier PEs is that it addresses these drawbacks to a degree that makes its use feasible. The following two drawbacks are the most important. First, for languages that use a textual syntax, users expect the editor to behave like regular, character-oriented text editors. Since PEs do not work with sequences of characters, this can be a challenge. According to the results of a not-yet-published survey, most

[4] `http://jetbrains.net/mps`
[5] `http://codeorchestra.com/ide/`

of our users (programmers and non-programmers) are happy with the editor after a few days of getting used to it. MPS addresses this challenge with a variety of approaches discussed in [10]; we mention two examples here. MPS supports *linear editing* of expressions such as `2+3` instead of requiring to first enter `+` and then the two arguments. Even though no parser is used because every token is bound immediately when entered, precedence, specified declaratively as a number for each operator, is taken into account. MPS also supports *cross-tree editing*; for example, parentheses can be entered in arbitrary locations to change, for example, `2+3*4` to `(2+3)*4`. Both, linear editing and cross-tree editing were not supported in earlier PEs.

The second challenge of PEs is infrastructure integration. PEs do not store programs as text, because this would re-introduce parsing and hence negate the advantages. Instead, the AST is persisted, typically as XML. For use in practice, the integration of these XML files with version control systems must be addressed: diff/merge must be supported using the concrete, projected syntax. MPS supports this, and is used routinely with `git` or `svn`.

The remainder of this article uses real-world languages from three different domains to illustrate how modular languages and projectional editing are used in (programming) language design. For each example, we describe the context, the addressed challenges, the way modular languages and PEs are used to address these challenges as well as preliminary experiences and conclusions. The systems described in the examples have been built or supported by the authors of this article.

2 Example: Embedded Programming

Context Embedded software must respect constraints regarding code size, memory or timing. At the same time, quality, maintainability and safety is critical. More and more devices contain software, making its importance grow steadily.

Challenges Much of today's embedded software is developed in C, because C code can be optimized manually to meet the constraints mentioned above. But C lacks the means to build new abstractions effectively, potentially hampering quality, maintainability and safety. Since new abstractions must imply little or no runtime overhead, the preprocessor is often used to build them. However, preprocessor-based abstractions are brittle, because the type checker, the IDE and static analysis tools only have limited awareness of these abstractions. In the following we discuss physical units and state machines to illustrate this problem.

Embedded software often works with real-world quantities, and annotating types and literals with units can enhance type safety by detecting problems like $\texttt{double/*metersPerSec*/ speed} = \frac{\texttt{10/*s*/}}{\texttt{5/*m*/}}$. C does not support annotating physical units to types and literals in a way that the type checker can detect such problems. Macros can reasonably be used for conversions (e.g., meters to feet, `m_to_ft(val)`). However, neither the IDE nor the compiler knows about the semantics of these macros and cannot check whether they are used correctly.

State machines can also be implemented using macros. In addition to being brittle for the reasons mentioned above, the result is also hard to read because of the limited syntactic flexibility of C macros. In practice, state machines are often implemented in plain C (using `switch` statements or cross-indexed arrays and function pointers) or with an external state machine modeling tool that generates C code. Both solutions are problematic. In the first case, the semantics of state machines is lost for the developer, the type checker, compiler and the IDE; it cannot easily be analysed for dead states or non-deterministic transitions. In the second case, integration is an issue: the modeling tool usually does not know the rest of the C-based system, and hence cannot check for the wrong use of variables or functions in a state machine.

Solution Approach mbeddr[6] incrementally adds abstractions for embedded software development to a modularized version of C. In addition to physical units and state machines (discussed below), mbeddr supports interfaces and components, unit testing as well as product line variability, requirements tracing and documentation. All of these abstractions are modular C extensions, so users can always fall back to C if the higher level abstractions are not efficient enough. Users can also build new extensions or create new generators for existing extensions. IDE support such as type checking, code completion, find usages and refactoring works seamlessly across language extensions. Debugging of extensions is supported with an extensible debugger architecture.

One of mbeddr's extensions is state machines, represented textually. State machines contain local variables, event declarations as well as states with entry/exit actions and transitions with guards and actions. Events are used to communicate with a state machine's environment and can be bound to different triggers (e.g., an in event can be bound to an interrupt; an out event to a function call). The default code generator for state machines generates a `switch` statement. However, like every generator in mbeddr, it can be exchanged with an optimizing generator for specific kinds of state machines or target platforms. Since state machine are represented first-class, they can be model-checked: mbeddr supports detection of dead states and non-deterministic transitions, and additional constraints can be defined using temporal logic. Failed properties are reported in terms of the state machine, not in terms of the generated code.

Another extension is the ability to annotate types and literals with units. The units are integrated with the C type system: an error is flagged in the IDE, if, for example, a variable that has the unit `m/s` is assigned an expression whose computed unit is different. The annotated units only affect the type system, so no runtime overhead is incurred. To convert units, conversion rules can be used. These are type safe in terms of C types as well as units.

Projection and Modularity mbeddr consists of 74 tightly integrated languages, most of them extensions to C. This modularity has several advantages.

[6] http://www.mbeddr.com

From a language developer's perspective the complexity of the individual languages is reduced, allowing relatively independent evolution of the languages. Users can choose which extensions to use in a program so they are not overwhelmed by a huge, monolithic language. In addition, users can incrementally grow mbeddr towards their domain by creating new, domain-specific extensions.

mbeddr's C extensions are not just coarse grained extensions that could easily be implemented with escapes in parser-based systems. In particular the units extend the very fine grained type and expression syntax. The ability to combine extensions – for example, state machines may use units in the guards – emphasize this.

MPS' projectional editor allows mbeddr to show the same model in different ways. For example, a state machine can alternatively be edited as a table, with events as the column header and states as row headers. The content cells contain the transitions for a given state/event combination.

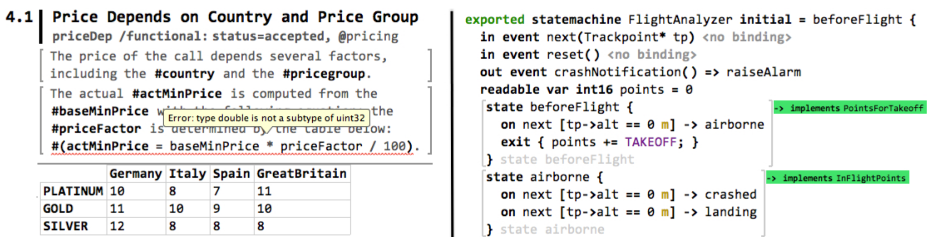
Experience An industrial project currently develops a Smartmeter, with hard real-time requirements and a memory-constrained target platform. Experience from this project shows that mbeddr's abstractions lead to more maintainable and testable software while at the same time not exceeding the resources available on the target hardware. Siemens PL (LMS) has selected mbeddr as the basis of their new controls engineering tool. Among other things, support for graphical data flow models and tabular data dictionaries is being added to mbeddr. The experience with mbeddr so far is summarized in [11].

3 Example: Requirements Engineering

Context Requirements are usually expressed as prose, plus some structured data such as tables. There is tool support beyond Word or Excel, exemplified by DOORS[7]. But even in DOORS, requirements are mostly expressed as prose. Many development processes and industry standards mandate tracing of requirements, where implementation artifacts are connected to the requirements that drive the particular implementation artifact. This aids maintainability, because the impact of a requirement change can be connected to potentially affected parts of the implementation.

Challenges Prose is processable only by humans. In addition, it is likely that the developers who read the requirements misunderstand some of them and implement the wrong functionality. One approach to address this problem, is to use controlled natural language, trying to be extremely precise in writing prose. Another approach is to express those requirements that *can be* formalized or structured with suitable machine processable languages. However, close integration between prose and such a diverse and growing set of domain specific languages is necessary, since some (parts of) requirements will always be expressed as prose.

[7] `http://www-03.ibm.com/software/products/en/ratidoor`



```
4.1 | Price Depends on Country and Price Group
    | priceDep /functional: status=accepted, @pricing
  [ The price of the call depends several factors,
    including the #country and the #pricegroup. ]
  [ The actual #actMinPrice is computed from the
    #baseMinPrice w[illegible] the
    #priceFactor is determined by the table below:
    #(actMinPrice = baseMinPrice * priceFactor / 100). ]
```

Error: type double is not a subtype of uint32

	Germany	Italy	Spain	GreatBritain
PLATINUM	10	8	7	11
GOLD	11	10	9	10
SILVER	12	8	8	8

```
exported statemachine FlightAnalyzer initial = beforeFlight {
  in event next(Trackpoint* tp) <no binding>
  in event reset() <no binding>
  out event crashNotification() => raiseAlarm
  readable var int16 points = 0
  state beforeFlight {                              -> implements PointsForTakeoff
    on next [tp->alt == 0 m] -> airborne
    exit { points += TAKEOFF; }
  } state beforeFlight
  state airborne {                                  -> implements InFlightPoints
    on next [tp->alt == 0 m] -> crashed
    on next [tp->alt == 0 m] -> landing
  } state airborne
```

Fig. 1. Left: Example requirement whose prose description contains variable definitions and a formula. As an example extension, the requirement also contains a pricing table. The variables, the formula, and the table are program elements and not just formatted text. **Right:** An example of traces attached to program code. Such traces can be attached to programs expressed in any language.

Creating complete and consistent traces is a lot of work and requires discipline, but it also requires tool support: it must be possible to attach traces to arbitrary program elements, expressed in any implementation language. Most of today's development tools do not support this.

Solution Approach mbeddr comes with a requirements language [12]. It represents each requirement with a unique ID, a summary, and a prose description. In addition, it allows embedding code expressed in any language into requirements, with full IDE support for those languages. Requirements elicitation can start with prose, and as the understanding grows, some requirements can be formalized with DSLs. For example, the left side of Fig. 1 contains a table used for price calculation in a hypothetical telecoms company. The table is not just formatting: countries and price groups are references to variables defined elsewhere. To further support the integrating prose and formal aspects, arbitrary program nodes can be embedded into prose [13]. The example embeds variable definitions, which are automatically renamed during refactorings. It also embeds price calculation formulas. The formulas are real, type checked expressions. During implementation, Java or C code can directly reference variables, formulas, or the pricing table. Code generators translate these formal descriptions into executable code.

The right side of Fig. 1 shows an mbeddr state machine. Some states have traces that point to requirements. Traces can be attached to program elements expressed in any language. Also, since the traces are actual *pointers* to requirements, they can be followed in reverse. mbeddr supports reports that show which requirement is traced from which program elements.

Projection and Modularity Language modularity is crucial for the approach. The basic requirements language that describes each requirement with prose, a summary, and an ID is generic and reusable. For particular domains, DSLs can be developed and plugged into the requirements language seamlessly. As illustrated by the above mbeddr C examples, even these DSLs can be extended further;

for example, an expression language that can be embedded in various kinds of business rules is a useful reusable asset.

Tracing also relies on language composition and projectional editing by using MPS' annotations. These are special kinds of nodes that can be attached to arbitrary program nodes, without them (or their definitions) being aware of that. This is useful for ”metadata” that is used by specialized tools (e.g., a trace analyzer), but does not affect the semantics of the core program. Documentation, requirements traces, or specification of architectural layers are examples.

The benefits of projectional editing are also exploited in the requirements language. The ability to seamlessly mix unstructured prose with structured program nodes is extremely helpful for descriptions in which prose has played, and will probably always play, an important role (see [13] for details). Requirements engineers and domain experts can start with prose and then enrich the prose itself with formal aspects such as references to domain entities, embedded formulas, or product specific value assignments to variables in the context of product lines. New ”embeddable words” can be defined at any time.

The ability to use notations that are not typically associated with languages, such as tables or mathematical formulas, is extremely useful in requirements engineering, where users are typically not programmers. Projectional editing is a good fit, because it can support such notations.

Experience mbeddr had been started as a collection of extensions of C, as discussed in the previous section. When MPS got support for mixing prose and program nodes (through a plugin by Sascha Lisson[8]), we realized MPS' potential for requirements engineering as well. The requirements language has been used in several requirements elicitation projects and has received very good feedback from its users.

4 Example: Insurance Rules

Context In the insurance industry new products are defined on a regular basis. The definition includes many, often complex, mathematical formulas to calculate values for premiums, annuities, reserves or dividends. These formulas play an important rule in the success of an insurance company. For example, if the premium is calculated too low, the company might lose money, while a premium that is too high might hurt sales. Time-to-market also plays an important role: companies must respond quickly to market opportunities or changes in the law by offering tailored insurance products. The formulas are typically written and maintained by actuaries. Actuaries are not programmers, they are mathematical insurance experts.

Challenges There are two common ways for implementing the insurance formulas today. In the first one, actuaries write the formulas in an informal language, and hand this specification over to programmers for implementation (for

[8] `https://github.com/slisson/mps-richtext`

example in Java). There are well-known problems with this approach: errors may be introduced in the communication between the involved people, and development speed is low because of the manual, multi-step process. The alternative is that the actuaries write their formulas in a formal language with downstream code generators, without involving programmers. The formal languages used in practice are essentially simplified programming languages. To use such languages, actuaries must, to a degree, become programmers, making this approach a tough sell.

The challenge is to define a language that is formal enough for code generation, while not alienating or overwhelming actuaries. The example below shows an early attempt at such a language that uses a *conditional assignment* based on a notation inspired by programming languages. It was rejected categorically because it was perceived to apply general programming syntax to a specialized insurance problem.

```
CASE
  WHEN CATV equals D1
    anui = ( SUM ( i , 1 , k , anui * 6 / prs + prd * ( i / 3 + 12 ) ) + 42 ) * arb
  WHEN CATV equals D2
    anui = SUM ( i , 1 , arb , INFWP [ i ] + local )
  WHEN true
    anui = SUM ( i , 1 , 12 , cal [ 1 ] + local )
ENDCASE
```

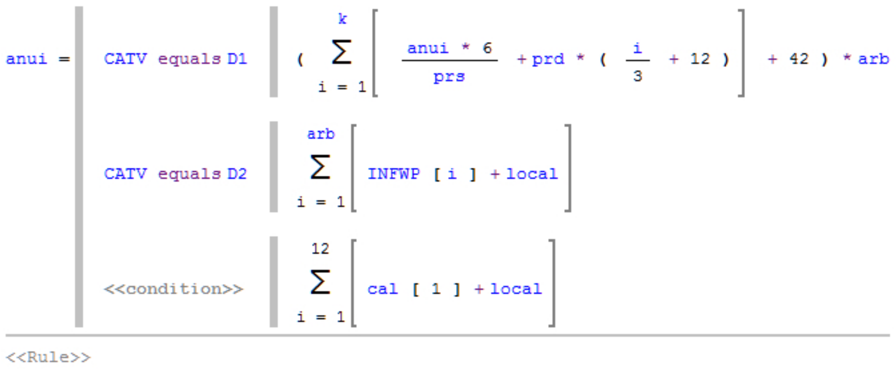
Solution Approach Fig. 2 shows the same conditional assignment with a notation that uses a column layout instead of keywords. The actuaries found this notation easy to grasp and accepted it without problems. A more easily understandable notation was also needed for complex formulas like the one from the first `when` clause in the above code:

```
( SUM ( i , 1 , k , anui * 6 / prs + prd * ( i / 3 + 12 ) ) + 42 ) * arb
```

While easy to parse for a computer, the linear structure is hard to "parse" for humans. We decided to use a non-linear, mathematical notation similar to what actuaries use on paper. The first line in Fig. 2 shows the same formula. The notation makes the structure immediately clear to users.

Non-programmers want the computer to indicate what must be done next. Therefore, we use placeholders (such as `«condition»` in Fig. 2) to indicate locations in the code where data can be entered. This is perceived to be easier than an empty editor that requires the user to figure out what is allowed next, possibly via code completion. Partial projections, configured via global settings, allow users to hide parts or aspects of programs, helping to focus on the task at hand. One such aspect is debug information. Since expressions have no side-effects, debugging does not require *stepping though* the program; the computation can be illustrated just by showing all intermediate results (see Fig. 3). It turned out that "test-driven rule development", and the ability to "overlay" the test data over the insurance rules, is very helpful for actuaries.

Projection and Modularity Projectional editing is essential for this system. Non-linear notations such as sum symbols, fraction bars, the column layout or



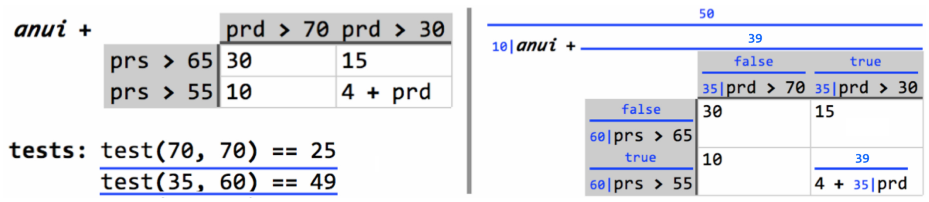
```
anui =  | CATV equals D1 | ( Σ_{i = 1}^{k} [ (anui * 6) / prs + prd * ( i / 3 + 12 ) ] + 42 ) * arb
        | CATV equals D2 | Σ_{i = 1}^{arb} [ INFWP [ i ] + local ]
        | <<condition>>  | Σ_{i = 1}^{12} [ cal [ 1 ] + local ]
<<Rule>>
```

Fig. 2. Domain specific insurance language using column layout, placeholders and mathematical notation. Using some graphical elements and placeholders helped with the acceptance of the language by end users.

tables are not practical with parsers. MPS can also use buttons, check boxes or labels and text fields in an editor. This allows mixing language-like notations (expressions, statements, math symbols) with UI elements known to the users from form-like applications, further lowering the adoption barrier.

The debug notation shown in Fig. 3 also illustrates the power of projectional editing. The debug information is shown only when the debugger is activated. It is read-only (computed by an interpreter) and updated automatically as test data changes. The tree-like notation that shows the value of each subexpression is essentially an automatic side effect of the tree structure of expressions.

Experience The insurance language is currently being developed. During the sessions with actuaries we received enthusiastic feedback about the notation, and they came up up with many new ideas to extend the notation with more insurance-specific symbols and structures. This experience is in stark contrast to earlier attempts based on more classical programming language notations and



anui +

	prd > 70	prd > 30
prs > 65	30	15
prs > 55	10	4 + prd

```
tests: test(70, 70) == 25
       test(35, 60) == 49
```

Fig. 3. Left, Top: An expression used in insurance rules using a decision table, a compact notation for nested `if` statements. **Left, Bottom:** Two test cases; the first argument is `prd`, the second one is `prs`. **Right:** The same expression in debug mode, where intermediate result of every subexpression (computed by an interpreter for a given test case) are annotated over or to the left of the expression.

tools. In the future, more aspects of the overall insurance product development workflow will be implemented for the customer, integrating the formulas discussed above. MPS' support for language composition makes this possible with very limited effort.

5 Summary

Based on our experience and feedback from users in domains as diverse as embedded software engineering, requirements management and insurance software, we conclude that the support for non-textual notations and wide-ranging modularity provided by projectional editing has significant advantages compared to "classical" languages. It enables the use of languages for tasks where languages have not been feasible before.

However, the *development* of such applications also needs evaluation. While beyond the scope of this article, our experience is that MPS is a very productive language workbench, as measured in the size of language implementations and effort spent (cf. [7]). For example, the state machines extension to mbeddr C was built in roughly 2 person months. Both MPS and mbeddr are open source; you can try for yourself.

Finally, it is not *absolutely* clear that systems such as the ones illustrated in this article can *only* be built with projectional editors. Perhaps parsing could be extended to support the necessary features. However, no such systems built with parser technology exist today. More systematic research is needed in this direction.

References

1. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, ACM (2012) 7
2. Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages? (2005)
3. Reps, T.W., Teitelbaum, T.: The Synthesizer Generator. In: First ACM SIGSOFT-/SIGPLAN software engineering symposium on Practical software development environments, ACM (1984)
4. Klint, P.: A Meta-Environment for Generating Programming Environments. ACM Transactions on Software Engineering Methodology **2**(2) (1993)
5. Klint, P., Van Der Storm, T., Vinju, J.: Easy meta-programming with rascal. In: Generative and Transformational Techniques in Software Engineering III. Springer (2011) 222–289
6. Kats, L.C., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and ides. In: ACM Sigplan Notices. Volume 45., ACM (2010) 444–463
7. Erdweg, S., Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., Vlist, K.,

Wachsmuth, G., Woning, J.: The State of the Art in Language Workbenches. In Erwig, M., Paige, R., Wyk, E., eds.: Software Language Engineering. Volume 8225 of Lecture Notes in Computer Science. Springer International Publishing (2013) 197–217

8. Simi, M., Campagne, F.: Composable Languages for Bioinformatics: The NYoSh experiment. PeerJ PrePrints **1:e112v2** (2013)
9. Kats, L.C.L., Visser, E., Wachsmuth, G.: Pure and declarative syntax definition: paradise lost and regained. In: Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, Reno/Tahoe, Nevada, ACM (2010)
10. 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
11. Voelter, M.: Preliminary experience of using mbeddr for developing embedded software. In: Proceedings of the 10th Dagstuhl Workshop on Model-based Development of Embedded Systems. (2014) 10
12. Voelter, M., Ratiu, D., Tomassetti, F.: Requirements as First-Class Citizens: Integrating Requirements closely with Implementation Artifacts. In: ACESMB@MoDELS. (2013)
13. Voelter, M.: Integrating Prose as First-Class Citizens with Models and Code. In: 7th International Workshop on Multi-Paradigm Modeling MPM 2013. (2013) 17