

Textual DSLs



www.mdsd-buch.de



www.mdsd-book.org

Markus Völter

voelter@acm.org
www.voelter.de



Together with
Sven Efftinge
and Arno Haase



Characteristics of Textual DSLs

- Different **characteristics** for different domains
 - **Concrete Syntax:** textual vs. graphical
 - **Domain Selection:** structural vs. behavioural
 - **Expressive Power:** declarative vs. imperative
 - **Execution:** interpretation vs. compilation/transformation
 - **Integration:** internal vs. external
 - **Tool Support:** editor, debugger, ...



Why concrete syntax is important

- Abstract Syntax defines grammar for the language – **most important for tools**
- Concrete Syntax is the “UI” for the language – **critical for DSL users**
 - concise vs. redundant
 - intuitive
 - simple to write and read
- Tool support matters
 - IDE integration
 - syntax highlighting
 - metamodel awareness



Different syntax for different abstractions / domains

- Different concrete syntax is **well established** for different domains
 - class diagrams to describe types and structure
 - state charts
 - textual expression notation for behaviour
 - XML for structured data
- The **abstraction should drive syntax** decision, not vice versa
 - Available tool support often decides the syntax
 - UML has its uses, but it is no panacea
 - building specific textual languages – and IDEs to work with them – has become feasible



Tradeoffs for textual DSLs

- With **both** textual and graphical syntax you can
 - Model for any meta model
 - verify constraints in real time
 - (Eclipse) write ordinary EMF models
- **Graphical Editors** are good to show structural relationships
- **Textual Editors** are better for „algorithmic“ aspects, or hierarchical models
- **Textual Editors** integrate better with CVS etc. (diff, merge)

Parts of a textual language (1): Parser

- Every textual language requires a **parser**
 - transform one or more text files into an **object representation** (aka AST)
 - check for **syntactical correctness**
 - Acts as a **linker**
- Parsers are typically **generated** based on a formal syntax definition (such as EBNF)
 - antlr, javacc, LPG, ...

Parts of a textual language (2): Processor

- There must be code that **“executes”** the language
 - **Generator:** generate code for a different language, typically at a lower level of abstraction
 - **Compiler:** generate byte code / machine code
 - **Interpreter:** walk through the object graph, executing code step by step
- **Interpreters** are especially suited for executing behavior,
- **Generators** are better at handling structural models and generating lower-level representations of these structures.
- Since textual DSLs are especially suited for behavioral descriptions, textual DSLs are often used together with interpreters.

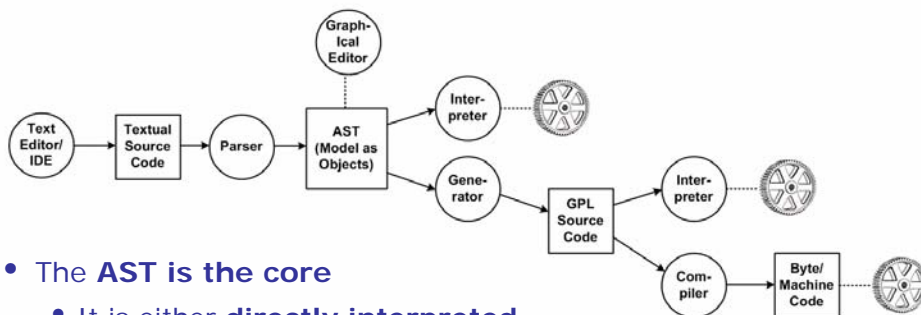
Parts of a textual language (3): IDE integration

- Developers have become used to good **tool support**
 - syntax highlighting
 - context sensitive assistance (here: metamodel aware code completion)
 - overview, cross-referencing
- These features are not strictly part of the language, but **very important** for user acceptance
 - One reason to use UML and / or XML
 - is a prerequisite to choosing based on the best fit for a domain

External vs. Internal DSLs

- **External DSLs** are defined **separately** from an existing language
 - separate parser and execution environment
 - separate editor / IDE
 - a whole separate language
 - No symbolic integration
- **Internal DSLs** are defined **within** an existing language
 - leveraging extension mechanism of the language – C macros, Ruby metaprogramming, Lisp macros, ...
 - good symbolic integration with the language
- **Internal DSLs** are also usually **limited**
 - syntactically constrained to what the host language offers
 - Typically not very good specific IDE support

External DSLs: Implementation



- The **AST is the core**
 - It is either **directly interpreted**
 - Or **transformed** into GPL code (which is then interpreted or further transformed, i.e. compiled)
- AST can be **created** by
 - **Direct editing** (typically via a graphical editor)
 - Or **via a parser** from a typically textual representation

Examples:
oAW, GMF, xText, etc.
(„classic“ MDSD)

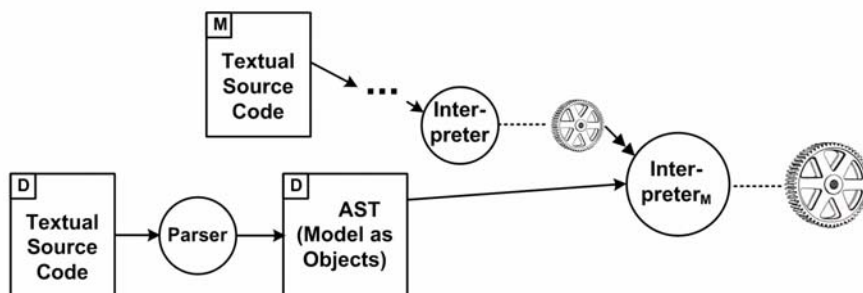
Internal DSLs: Interpreted I: Metaprogramming



- Source code contains the **metaprogram** (M) defining the DSL as well as a **program** in the DSL (D)
- After parsing, the **AST contains the metaprogram and the program** (this is possible, since D is syntactically compatible with the host language)
- In the interpreter, the **DSL program D uses the metaprogram M** and produces the desired effect

Examples:
Lisp, Ruby

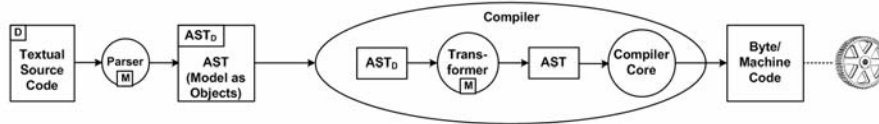
Internal DSLs: Interpreted I: Metaprogramming II



- The (often separate) **metaprogram M modifies the interpreter** effectively producing a custom interpreter that knows about M and can interpret DSL programs D
- The modified interpreter **interprets the DSL program D** as part of the host program

Example: CLOS

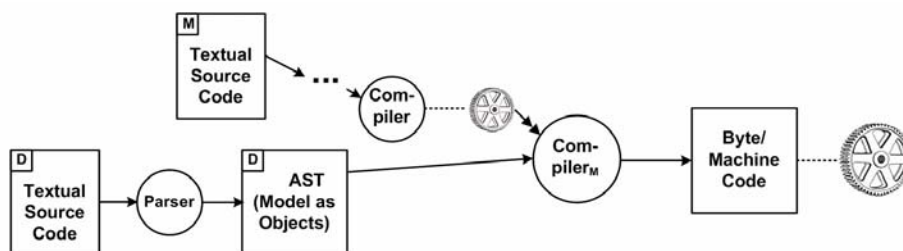
Internal DSLs: Compiled I



- The program contains **host code and DSL code D**.
- A **parser that knows about D** builds an AST with a part that is specific to M (AST_D).
- Inside the compiler, a **special transformer** (M-specific) transforms AST_D into a regular AST which then compiled with the compiler code of the host language.
- In **homogenous systems**, the language for implementing $Parser_M$ and $Transformer_M$ are the same as the host language (program and metaprograms can be mixed, too).

Example: Converge

Internal DSLs: Compiled II



- The **metaprogram modifies the Compiler** to understand D programs (aka open compilers, Compile-Time MOP)
- The **Compiler_M now understands D** – depending on how far the modification goes, D can have specific syntax or not
- In **homogenous systems**, the language for implementing M are the same as the host language (program and metaprograms can be mixed, too).

Example: OpenC++

Strengths of Textual DSLs

- Textual languages have **specific strengths** compared to graphical languages
 - ideally there should be the option to have both
- **compact and expressive syntax**
 - productivity for experienced users
 - IDE support softens learning curve
- **configuration management/versioning** and integration into the “regular” development process
 - splitting a model into several files
 - concurrent work on a model, especially with a version control system: diff, merge
 - search, replace



Tooling for Textual DSLs

- **oAW's xText**
 - Open Source (at Eclipse GMT)
 - Simple, starting with the concrete syntax definition
 - Allows for custom constraint integration
- **INRIA's TCS** (Textual Concrete Syntax)
 - Open Source (at Eclipse GMT)
 - Starts with the domain metamodel
 - Support for operators and serialization
- **Eclipse Modeling TMF is in preparation...**
- **IBM's SAFARI** (T.J. Watson)
 - Not open source (yet)
 - Very powerful framework for building Eclipse IDEs for custom languages – keep an eye on it!

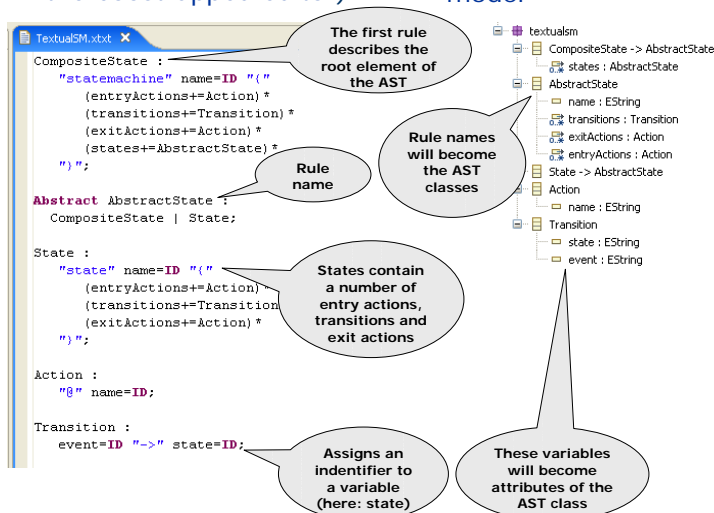


oAW xText (1)

- oAW's textual DSL generator framework xText simplifies defining a textual syntax for a domain-specific language
- Based on a **BNF-like syntax definition language** it generates:
 - An **EMF-based metamodel** (representing the AST)
 - An **Antlr** parser instantiating dynamic EMF-models
 - An **Eclipse text editor plugin** providing
 - syntax highlighting
 - customizable outline view,
 - Syntax folding
 - syntax checking
 - as well as constraints checking based on a Check file, as always oAW

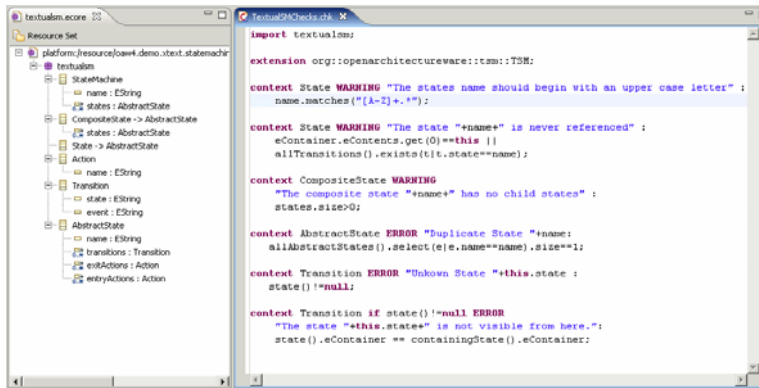
xText (2)

- The **grammar** (shown in the bootstrapped editor)
- The **generated eCore AST model**



xText (3)

- You can define **additional constraints** that should be validated in the generated editor.
- This is based on oAW's *Check* language
 - i.e. These are constraints like all the others you've already come across



xText (4)

- The **generated editor** and its **outline view**

