An extensible Validation Framework for DSLs using MontiCore

On the Example of Coding Guidelines

Christian Berger, Bernhard Rumpe, Steven Völkel
Lehrstuhl Informatik 3 (Software Engineering)
RWTH Aachen

http://www.se-rwth.de/
Motivation

- Syntactic quality of code can help to improve the development process as
  - it increases readability
  - and understandability
  - and thus can help to find/prevent errors

- Coding guidelines are often seen as „recommendations“, only sometimes tools are used

- Especially for C++ only a few tools exists (there are a lot for Java)

- Problems:
  - Each tool for each language has its own architecture (can we identify a common part?)
  - Tools are often not extensible/configurable
  - Tools are badly integrated with other parts: parser, lexer, abstract syntax, IDEs, code generators

- Therefore: An extensible Validation Framework for DSLs using MontiCore
## Content

1. Basic MontiCore
2. Concepts of Modularity
3. Language Inheritance
4. Language Embedding
5. Beyond Syntax
6. The Validation Framework
7. Example: C++ Guidelines
8. Conclusion
Short Overview: MontiCore

- Framework for development of compositional textual DSLs
- Grammar based, similar to common parser generator formats
- Integrated definition of abstract & concrete syntax
- Input: extended grammar format with meta-modeling concepts
  - Rule inheritance
  - Interfaces
  - Associations
- Output: parser, lexer, abstract syntax classes, symbol tables, ...
- Main target: Compositionality
MontiCore – Creation of domain specific program synthesizers (DSLTools)

- MontiCore can be used to create domain specific program synthesizers

* The components are organized in libraries and are created by generation or coding
The MontiCore Grammar Format

- Grammars have a name and a package like Java – permits references to grammars

- No explicit start rule, can use every rule to start

- Regular lexer rules
  - possibility to define a mapping to Java types

- Context-free parser rules
  - syntactic and semantic predicates, because we use ANTLR to generate lexers and parsers
  - LHS: name of the parser rule
  - RHS: combination of keywords, references to other nonterminals/terminals – including alternatives ( | ), blocks, optional elements ( ? ), unbound repetitions ( * ), etc.

- Some other features: explained in this talk
Running Example

- Example:
  - Bookstore: has a name, consists of books and journals
  - Books: have an id, a title, and authors (persons)
  - Journals: have an id and a title
  - Persons: have a forename and a lastname
package mc.examples.bookstore;

grammar Bookstore {

    // Create a token "ID" which is reflected
    // as int in the abstract syntax
    ident ID ('0'..'9')+ : int;

    Bookstore = "bookstore" name:IDENT
        "{" ( Book | Journal )* "}" ;

    Book = "book" id:ID title:STRING "by"
        authors:Person ("," authors:Person)* ";";

    Journal = "journal" id:ID title:STRING ";" ;

    Person = forename:IDENT lastname:IDENT ;
}

The Basic MontiCore Grammar Format

```
package mc.examples.bookstore;

grammar Bookstore {

    // Create a token "ID" which is reflected
    // as int in the abstract syntax
    ident ID ('0'..'9')+ : int;

    Bookstore = "bookstore" name:IDENT
                 "{" ( Book | Journal )* "}" ;

    Book = "book" id:ID title:STRING "by"
           authors:Person ("," authors:Person)* ";";

    Journal = "journal" id:ID title:STRING ";" ;

    Person = forename:IDENT lastname:IDENT ;
}
```
package mc.examples.bookstore;

grammar Bookstore {

    // Create a token "ID" which is reflected
    // as int in the abstract syntax
    ident ID ('0'..'9')+ : int;

    Bookstore = "bookstore" name:IDENT
                "{" ( Book | Journal )* "}" ;

    Book = "book" id:ID title:STRING "by"
         authors:Person ("," authors:Person)* ";";

    Journal = "journal" id:ID title:STRING ";" ;

    Person = forename:IDENT lastname:IDENT ;
}

Terminals result in attributes

BookStore
  Name:String
    Book
      Title:String
      Id: int

  *
The Basic MontiCore Grammar Format

```java
package mc.examples.bookstore;

grammar Bookstore {

    // Create a token "ID" which is reflected
    // as int in the abstract syntax
    ident ID ('0'..'9')+: int;

    Bookstore = "bookstore" name:IDENT 
        "{" ( Book | Journal )* "}" ;

    Book = "book" id:ID title:STRING "by" 
        authors:Person ("," authors:Person)* ";";

    Journal = "journal" id:ID title:STRING ";" ;

    Person = forename:IDENT lastname:IDENT ;
}
```

Refs. to nonterminals result in compositions

Cardinality (other cardinalities can be specified)
The MontiCore Grammar Format

More on syntax:

- Can specify associations, resolvers are generated automatically or can be hand-coded

- Abstract syntax classes contain:
  - Tree-traversal-methods (Visitor)
  - Get- and set-methods
  - Clone-methods
  - ...

- Can specify additional
  - Attributes
  - Methods
  - Are woven into abstract syntax classes

- ...

Concepts of Modularity

- Typical example: UML + OCL

- 3 Observations:
  1. One can identify a “core” UML
     - Classes, stereotypes, …
     - Reasonable to extract those elements and reuse them in all languages
  2. One could imagine to use
     another constraint language than OCL
  3. One could combine OCL with
     other languages than UML

MontiCore:
  - Language Inheritance: point 1
  - Language Embedding: point 2 + 3

Requires loose coupling
## Content

| 1. | Basic MontiCore |
| 2. | Concepts of Modularity |
| 3. | Language Inheritance |
| 4. | Language Embedding |
| 5. | Beyond Syntax |
| 6. | The Validation Framework |
| 7. | Example: C++ Guidelines |
| 8. | Conclusion |
Language Inheritance

- Multiple inheritance: one grammar can extend one or more supergrammars

- Add new productions / override rules / add alternatives…

- All elements of supergrammars are visible: specification of the delta only

- Example: Adding editor information to journals
Language Inheritance

- Refining journals (old version had no editors):

```java
package mc.examples.bookstore2;

grammar ExtendedBookstore extends mc.examples.bookstore.Bookstore {

    Journal = "journal" id:ID title:STRING "editors"
    editors:Person ("," editors:Person)* ";" ;

}
```

**Grammar inheritance**

**Override rule of supergrammar (same name)**
Language Inheritance

- Effects on abstract syntax:
  - Comparable to UML package merge, but
  - Overridden rules lead to oo-inheritance
  - Classes for non-overridden rules are not regenerated
Language Inheritance

There will be no instance of the old version!
Language Inheritance

- Effects on abstract syntax:
  - Comparable to UML package merge, but
  - Overridden rules lead to oo-inheritance
  - Classes for non-overridden rules are not regenerated

- Why?
  - Reuse
  - Algorithms which worked for old version may still work for the new one (minor modifications for overridden rules)
  - E.g. pretty printing, transformations, code generation, symbol tables …
Language Inheritance

- This version applies to situations where the designer of the old version does not foresee changes.
- One could imagine to add audio books.
- Other solution: introduce interfaces.

```plaintext
interface Item;

ast Item =
  title:STRING;

Bookstore = "bookstore" name:IDENT
  "{" (items:Item)* "}";

Book implements Item =
  "book" id:ID title:STRING "by"
  authors:Person
  ("," authors:Person)* ";";

Journal implements Item =
  "journal" id:ID title:STRING ";";
```

Introduce interface

All items have at least a title

Use interfaces just like nonterminals

This rule implements the interface
Language Inheritance: abstract syntax

```plaintext
Interface Item;

ast Item =
  title:STRING;

Bookstore = "bookstore" name:IDENT
  "{" (items:Item)* "}";

Book implements Item =
  "book" id:ID title:STRING "by"
  authors:Person
  ("," authors:Person)* ";";

Journal implements Item =
  "journal" id:ID title:STRING ";";
```
Language Inheritance

- Bookstore grammar is “open to extensions”
- New subgrammar: designer can simply add new items
- “implements” – relationship between new items and old interface

```
grammar ExtendedBookstore extends mc.examples.bookstore.Bookstore {
    AudioBook implements Item = "audiobook" id:ID title:STRING ";";
}
```

- Problem: both grammars are not decoupled
  - One could imagine to reuse audio books in a record shop
  - Then: change audiobook grammar (extends record shop grammar) – the new version understands book stores and record shops 😞
- Solution: Multiple inheritance
Language Inheritance

- New version:

```plaintext
grammar Audio {
    AudioBook = "audiobook" id:ID title:STRING ";";
}

grammar ExtendedBookstore extends mc.examples.bookstore.Bookstore,
    mc.examples.audiobook.Audio {
    AudioBook implements Item;
}
```

No need to repeat definition of AudioBook!
Language Inheritance

- Language inheritance is typically used when sublanguage is similar to superlanguage (add / remove something only)

- Problem: Lexical analysis
  - Subgrammar may have problems to understand instances of the supergrammar even when adding something only
  - Example: keywords
  - Known Example: introduction of “assert” in Java 1.4

- Solution: Decouple lexers / parsers by language embedding
Content

1. Basic MontiCore
2. Concepts of Modularity
3. Language Inheritance
4. Language Embedding
5. Beyond Syntax
6. The Validation Framework
7. Example: C++ Guidelines
8. Conclusion
Language Embedding

- DSLs are designed for a specific task => often necessary to combine several languages (e.g., OCL + CD + …)

- In this example: convenient to write OCL statement nearby the element it is constraining

- “Normal” approach: 1 monolithic grammar for OCL + CD. No reuse.

- Especially problematic when combining more than 2 languages

- Desirable: flexible mechanism
Language Embedding

- MontiCore grammars can be parameterized using an arbitrary number of external nonterminals
- Content of this external nonterminals not known at compilation time
- Languages can be combined at configuration time (each ext. NT seperately)
- Example here: use a description language for items (e.g., bibtext)

```plaintext
external Bookentry;
external Journalentry / example.IJournalEntry;

Book = "book" id:ID title:STRING
   "by" authors:Person ("," authors:Person)*
   Bookentry ";";

Journal = "journal" id:ID title:STRING Journalentry ";";
```
Language Embedding: Behind the Scenes

- Generate parsers / lexers independently
- Configure the combination (Java or DSL) – No need to recompile!
- MontiCore does the switching
- Results in the following structure:

```
Grammar bookstore
External Bookentry;
```

```
Grammar bibtex
BibtexBook = .....;
```

```
Concrete Lexer/Parser
```

```
Configuration:
bookstore.Bookentry -> bibtex.BibtexBook
```

```
Concrete Lexer/Parser
```

```
Superordinated Lexer/Parser
```

```
Grammar bookstore
```

```
Compiled independently
```

```
Concrete Lexer/Parser
```

```
Concrete Lexer/Parser
```

```
Superordinated Lexer/Parser
```
More on Language Embedding

- Configuration - DSL

// define rule Bookstore of the grammar
// mc.examples.bookstore.Bookstore
// to be used when starting to parse the text
mc.examples.bookstore.Bookstore.Bookstore bst <<start>>;

// embed bibtex rule for books as Bookentry

// embed bibtex rule for journals as Journalentry

- Even possible to switch language depending on input
| 1.   | Basic MontiCore          |
| 2.   | Concepts of Modularity   |
| 3.   | Language Inheritance     |
| 4.   | Language Embedding       |
| 5.   | Beyond Syntax            |
| 6.   | The Validation Framework |
| 7.   | Example: C++ Guidelines  |
| 8.   | Conclusion               |
Beyond Syntax

- Until now:
  - Parser, lexer generation
  - Abstract and concrete syntax definition
  - Modularity

- MontiCore offers
  - Standard means for error reporting
  - Standard means for file creation
  - Choice for code generation: own template language or by using printers
  - Standard means for tree traversal
  - Eclipse plugin generation
  - …

\[\begin{align*}
\text{Not specific for compositional development} \\
\text{Specific for compositional development}
\end{align*}\]
Beyond Syntax: Tree Traversal

- MontiCore tree traversal mechanism respects compositional syntax development

- Esp. useful for embedding

- Define visitors independently for each language
  - Bookstore
  - Bibtex

- Combine them at configuration time just as done with parsers/lexers

- Example: Pretty Printing
Beyond Syntax: Editor Generation

- MontiCore can generate Eclipse plugins: editors with
  - Syntaxhighlighting
  - Outlines
  - Selectable error messages with goto
  - Foldable code regions
  - ...

- Defined by additional information in the grammar

- Editor generation mechanism respects compositional syntax development

- Define information (keywords, which outline items…) independently for each language
  - Bookstore
  - Bibtex

- Combine them at configuration time just as done with parsers/lexers
Example:

```
keywords: bookstore, book, by, journal;

foldable: bookstore;

segment: bookstore ("pict/store.gif")
  show: "bookstore" Name ;
segment: book ("pict/book.gif")
  show: "book " Title "( " ID ")";
segment: journal ("pict/journal.gif")
  show: "journal " Title "( " ID ")";
```
Beyond Syntax: Editor Generation

Example

```java
book store Example{

// hot new books
book 1 "LNBIP No. 11" by
    Richard Paige,
    Bertrand Meyer;

book 2 "LNCS No. 5063" by
    Antonio Vallecillo,
    Jeff Gray,
    Alfonso Pierantonio;

journal 1 "Software and Systems";
}
```

Errors (1 item)

- expecting "bookstore", found 'book'
  Example,,... Bookstore line 1
Content

1. Basic MontiCore
2. Concepts of Modularity
3. Language Inheritance
4. Language Embedding
5. Beyond Syntax
6. The Validation Framework
7. Example: C++ Guidelines
8. Conclusion
The Validation Framework

- Developed in a project with Volkswagen

- Language-independent
  - Thus usable for GPLs and DSLs
  - Here: C++
  - Supports compositionality as it is based on the MontiCore framework

- Based on abstract syntaxes generated by MontiCore

- Uses different features of the MontiCore framework, e.g.
  - AST traversal
  - File generation
  - Error/violation reporting
  - …
Design considerations

- Said before: language independent
  - Therefore: must not depend on a concrete syntax
  - But: can be extended for a special language

- Visitor-Pattern for AST-traversal can be used
  - But one visitor for each coding guideline is too slow
  - Therefore: One visitor for all nodes, register at this visitor

- Configure easily which guidelines should (not) be used
  - Without touching code
  - Thus: configuration file

- Possibility to use it in continuous integration systems

- Output should be usable for different applications
  - Browsers
  - Email-notification
  - Database…
General Architecture

Output (can be serialized to XML)

1 per Guideline

Provided by Monticore

Generic Visitor (language independent)
Framework Workflow

1. Read configuration
2. Generate ASG from input file
3. Traverse ASG using the language dependent visitor
4. If [Has next matching VR], repeat step 3
5. If [No matching VR], go to step 6
6. If [Has nonterminal], repeat step 3
7. If [No more nonterminals], go to step 8
8. Output results
9. Report messages
10. Apply next VR

Flowchart:
- Start: Read configuration
- Read configuration → Generate ASG from input file
- Generate ASG from input file → Traverse ASG using the language dependent visitor
- Traverse ASG using the language dependent visitor → Decision: [Has next matching VR]?
  - Yes: Traverse ASG using the language dependent visitor
  - No: Decision: [No matching VR]?
    - Yes: Traverse ASG using the language dependent visitor
    - No: Decision: [Has nonterminal]?
      - Yes: Traverse ASG using the language dependent visitor
      - No: Output results
- Output results → Report messages
- Report messages → Apply next VR
- Apply next VR → Decision: [Has next matching VR]?
Application in an Automotive Project

- Automotive Project with Volkswagen
  - For automatically driving vehicles

- Implemented well-known Motor Industry Software Reliability Association (MISRA)-rules

- Aims:
  - Automatically apply MISRA-rules on the source code
  - Integrate it into continuous integration system
  - Check rules after each commit
  - Report results in a web-based framework
The C++ Grammar

- Regular compiling process starts with preprocessing
  - Resolves all preprocessor directives
  - `#include`-statements are embedded directly into source files

- Our tool starts after preprocessing stage

- Problem: C++ is not LL(k) or LR(k)

- Solution:
  - MontiCore is based on ANTLR and thus permits syntactic and semantic predicates
  - Syntactic predicates: look into „future“
  - Semantic predicates: insert code to decide between alternatives
    - Here: use of symbol table

- Result: 155 rules in 2600 LOC
MISRA-rules

- 3 classes of rules: SHOULD, SHALL, WILL

- First two are obligatory, last allows exceptions

- Which rules to check can be defined in a config file

- LOC for a rule:
  - Simple (e.g., naming conventions): 5
  - Complex (e.g., all public methods of a class should be part of its interface): 35

- Some implemented examples:
  - See next slide
Example rules

- ConstructorChecker: Validation of a specific order of constructors, destructors and other methods
- DestructorChecker: Asserting a virtual destructor
- EnumChecker: Validation ISO-like enum-declarations
- FlowControlChecker: Checking for flow control (gotos, breaks, labels)
- FunctionChecker: Various validations for function and method usage (e.g., naming, maximum number of lines...)
- IfChecker: Checking for braces in if-statements
- InitializedVariableChecker: Checking the initialization of local variables
- InterfaceChecker: Ensuring that all public methods from a class are declared in interfaces used by this class
- SingleLetterVariableChecker: Searching for variable names containing only one letter
- SwitchChecker: Checking for braces in switch-case-statements
Results: Trac using XSLT

CGL Report Summary

Overview:

CGL Categories

- Naming conventions checker (3 errors)
- Interface Checker (1 error)

Files:

- ExampleImpl.cpp (4 errors)

The CGLs (Found 4 errors; created 2009-03-29 14:45):

Naming Conventions Checker (3 errors):

Checks whether identifier naming is unique.
Reference: MISRA AV Rule 48
Priority: SHOULD

<table>
<thead>
<tr>
<th>Filename</th>
<th>Message</th>
<th>Row</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExampleImpl.cpp</td>
<td>Local Variable &quot;ll&quot; is named similar to &quot;ll : int&quot;.</td>
<td>15</td>
<td>7</td>
</tr>
<tr>
<td>ExampleImpl.cpp</td>
<td>Local Variable &quot;arraySize&quot; is named similar to &quot;array_Size : BOOL&quot;.</td>
<td>19</td>
<td>7</td>
</tr>
<tr>
<td>ExampleImpl.cpp</td>
<td>Local Variable &quot;tempInt&quot; is named similar to instance variable &quot;tempInt : INT&quot;.</td>
<td>22</td>
<td>7</td>
</tr>
</tbody>
</table>

Interface checker (1 errors):

Checks for correct interface usage.
Priority: SHALL
Configured properties:
ClosedAPI : true

<table>
<thead>
<tr>
<th>Filename</th>
<th>Message</th>
<th>Row</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExampleImpl.cpp</td>
<td>Public function &quot;derive : DOUBLE&quot; not derived from pure virtual method.</td>
<td>10</td>
<td>7</td>
</tr>
</tbody>
</table>
Result

- Checked about 100,000 LOC
- ~ 5,300 warnings found
- 8.6% highly critical warnings, e.g.
  - assignments in boolean expressions
  - misuse of memory handling
- 29.5% medium warnings, e.g.
  - Uninitialized variables
- 61.9% non-critical warnings, e.g.
  - Layout/design warnings
- All critical warnings have been reviewed (and fixed) by developers
Role of Compositionality for the Validation of C++ programs

- Can define C++ dialects by inheritance
  - Validation rules can still be applied
  - Maybe some new rules for new language features
  - Or remove some unnecessary using config file

- Reuse parts of C++
  - E.g. as action language in automata via embedding
  - Validation rules can still be used
  - Can define rules for automata as well
  - Both rule sets can be combined easily
Content

1. Basic MontiCore
2. Concepts of Modularity
3. Language Inheritance
4. Language Embedding
5. Beyond Syntax
6. The Validation Framework
7. Example: C++ Guidelines
8. Conclusion
Conclusion: MontiCore

- MontiCore supports compositional language development

  - Multiple Language inheritance:
    - Overriding / Adding rules
    - Usage of interfaces to design an open DSL for extensions in the future
    - Specify the delta only
    - Regenerate the delta only to enhance reuse

  - Language embedding:
    - Grammars may have external nonterminals
    - Can be constrained via handwritten interfaces
    - Combination at configuration time, no recompilation necessary
    - Language switching may depend on the input
Conclusion: MontiCore

- MontiCore supports not only compositional syntax definition
  - Standard means for common tasks like file handling or error reporting
  - Compositional tree traversal
  - Compositional editor with different comfort functionalities

- Use cases:
  - Version of UML (CDs, SDs, SC, ODs)
  - OCL
  - Java 5
  - DSL for view-based modeling of automotive architectures in industrial context
  - Bootstrapping
  - …
Conclusion: Validation Framework

- Language-independent framework based on MontiCore
- Can be used to check guidelines for DSLs and GPLs
- Extensible by writing own checks
- Configurable by config file
- Example: MISRA C++ guidelines in an automotive project
- Other use cases:
  - Sequence charts
  - Java
  - OCL
Thanks.

Q & A