Deriving Fluent Internal Domain-Specific Languages from Grammars

Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann
Software Engineering, RWTH Aachen University, Aachen, Germany
<lastname>@se-rwth.de

Abstract

A prime decision of engineering domain-specific languages (DSLs) is implementing these as external DSLs or internal DSLs. Agile language engineering benefits from easily switching between both shapes to provide rapidly developed prototypes before settling on a specific syntax. This switching, however, is rarely feasible due to the effort of re-implementing language tooling for both shapes. Current research in software language engineering focuses either on internal DSLs or external DSLs. We conceived a concept to automatically derive customizable internal DSLs from grammars that operate on the same abstract syntax as the external DSL. This supports reusing tooling (such as model checkers or code generators) between both shapes. We realized our concept with the MontiCore language workbench and Groovy as host language for internal DSLs. This concept is applicable to many grammar-based language definition formalisms and can facilitate agile language engineering.

CCS Concepts • **Software and its engineering → Software notations and tools**; **Domain specific languages**;

Keywords External DSLs, Internal DSLs, Grammarware

ACM Reference Format:
Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann. 2018. Deriving Fluent Internal Domain-Specific Languages from Grammars. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3276604.3276621

1 Introduction

Research and engineering have produced various approaches to develop textual domain-specific languages (DSLs). External textual DSLs [28, 31, 33] usually rely on grammars defining their syntax and parsers translating the DSLs' concrete syntaxes into internal representations, such as abstract syntax trees (ASTs). Furthermore, they typically comprise tailored infrastructures for model checking [15], transformation [17], editing [32], execution [18], and debugging [4]. Internal DSLs [9, 23, 27] rely on general-purpose programming languages (GPLs) as host languages and provide infrastructure (syntax, type system, editors, *etc.*) to the internal DSL, *i.e.,* an "internal DSL is just a particular idiom of writing code in the host GPL. So a Ruby internal DSL is Ruby code, just written in a particular style which gives a more language-like feel" [12]. Hence, GPLs with flexible syntax (*e.g.,* Groovy or Scala) are particularly well-suited.

Engineering a DSL requires a number of design decisions regarding syntax, semantics, and realization in technological spaces. Deciding on its shape is a prime design decision affecting many subsequent decisions, including syntactic flexibility, executability of its models, extensibility, and support by development tools [8]. Engineering internal DSLs can require significantly less effort than engineering external DSLs and they are favorable when the DSL users are already familiar with the host GPL [8]. In contrast, where a DSL requires domain-specific editor assistance or a specific syntax tailored towards non-software-experts, externals DSLs seem more appropriate [8]. Agile language engineering envisions short iteration cycles with the customers exploring language prototypes. However, rapidly prototyping an internal DSL and switching to an external DSL later (or vice versa) rarely is feasible due to the effort in re-engineering corresponding language realization constituents.

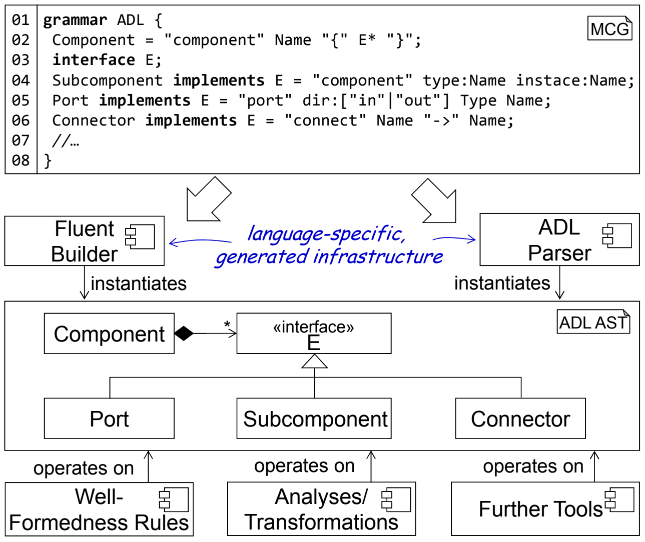
To facilitate agile switching between both shapes, we conceived a concept to automatically derive internal DSL tooling and external DSL tooling from the same grammar. Both DSL shapes use the same abstract syntax, but the external DSL instantiates it via parsers, while the internal DSL employs a generated builder infrastructure. This facilitates switching to an external DSL – if required at all – and prevents detaching both shapes. Further, it enables using tooling developed for the external DSL (*e.g.,* model checkers, code generators) with models constructed via the internal DSL, hence, enables "metamorphic" [1] (*i.e.,* multi-shape) DSLs. The internal DSL can be tailored to changing customer requirements easily without need to change related tooling (*e.g.,* editors, execution infrastructure) as dramatically as with external DSLs. This enables exploring the DSL with customers more flexibly

SLE '18, November 5–6, 2018, Boston, MA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6029-6/18/11...$15.00
https://doi.org/10.1145/3276604.3276621

and ultimately can reduce development effort. To this end, this paper contributes (1) a concept to derive fluent internal DSLs from external, grammar-based DSLs, and (2) a description of its realization with the MontiCore [16, 28] language workbench. In the following, Section 2 presents a motivating example before Section 3 discusses preliminaries. Section 4 describes our concept and Section 5 explains its realization. Section 6 presents a case study and Section 7 debates observations. Section 8 discusses related work. Section 9 concludes.

2 Example

Developing software languages usually is an iterative process. For instance, the design of the concrete syntax should be discussed in close relation with the users to integrate their requirements. This requires an efficient method to adapt the concrete syntax while implementing the tooling against a more stable abstract syntax. Consider the development of a small architecture modeling language (ADL). The grammar of this language is depicted in Figure 1 (top). The language should describe component models that comprise a name (l. 2) and subcomponents (l. 4), ports (l. 5), and connectors (l. 6). Ports are directed, typed interfaces for communication with other component models. Connectors connect ports to enable communication between these. Subcomponents are instances of other component models that enable hierarchical decomposition of components. Many language workbenches, such as MontiCore, use grammars as description for the language's syntax and generate language-processing infrastructure (*e.g.*, parser, abstract syntax classes) from it. With our approach, we instantiate the generated abstract syntax classes via a *fluent builder* resembling an internal DSL. This enables deriving the concrete syntax for an internal DSL operating on the same abstract syntax as the parser (*cf.* Figure 1 center). While the concrete syntax of the language can still be adjusted by integrating handwritten extensions to the fluent builder, the abstract syntax can be used to implement language tooling, such as well-formedness rules and other analyses and transformations (*cf.* Figure 1 bottom). The internal DSL yields the advantage that no parsing is necessary for instantiating the abstract syntax tree and that tooling (*e.g.*, editors) for the host language can be employed while these are not available yet for the external DSL. Additionally, the process of instantiating the abstract syntax tree is more transparent with fluent builders. This can be leveraged, *e.g.*, to investigate potential flaws regarding ambiguities in the instantiation of the abstract syntax. Even if only the internal DSL is to be used, our concept provides a good approach for describing its syntax. It generates complete tooling (*i.e.*, an abstract syntax data structure and the fluent builder for its instantiation) that can be adjusted with handwritten code from a grammar. This alleviates language engineers from implementing internal DSL tooling from scratch, which is



```
01 grammar ADL {
02  Component = "component" Name "{" E* "}";
03  interface E;
04  Subcomponent implements E = "component" type:Name instace:Name;
05  Port implements E = "port" dir:["in"|"out"] Type Name;
06  Connector implements E = "connect" Name "->" Name;
07  //…
08 }
```

Figure 1. Overview of the interplay between internal and external DSL of an exemplary ADL language.

usually error-prone with regard to checking the correct order of statements and to disabling undesired statements of the host language in internal DSL models.

3 Preliminaries

The realization of deriving internal DSLs from context-free grammars (CFGs) relies on the MontiCore language workbench and Groovy as host language for internal DSLs.

3.1 The MontiCore Language Workbench

MontiCore [21, 22, 28] is a language workbench [12] for developing external, textual DSLs. To define abstract syntax and concrete syntax of a language in an integrated fashion, MontiCore relies on an extended, customized notation of EBNF grammars [2]. From such grammars, MontiCore generates language tooling, including a parser, an AST data structure, and a visitor infrastructure for traversing AST instances. For well-formedness checks not expressible with CFGs, MontiCore supports well-formedness rules (context conditions) implemented as Java classes. These are checked against the AST by employing the generated visitor infrastructure. Analyses and transformations on the AST can be implemented via the visitors as well. The aforementioned constituents of MontiCore are depicted in Figure 2.

The syntax of MontiCore grammars borrows fundamental concepts of EBNF grammars and enriches these with concepts known from object-oriented programming [28]. The grammar language of MontiCore is bootstrapped. An excerpt of the metagrammar for MontiCore grammars optimized for human reading is depicted in Figure 3: Each grammar begins with the keyword grammar, followed by a name and, optionally, names of one or more supergrammars

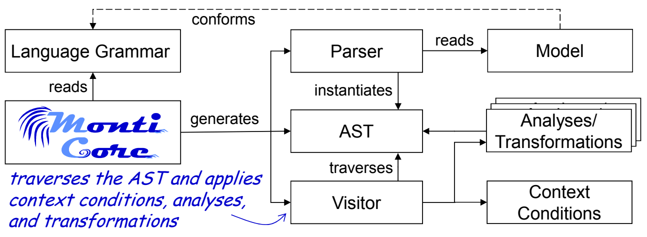


Figure 2. Overview of MontiCore language constituents.

```
01 grammar MC {                                                  MCG
02  GrammarModel = "grammar" Name Ext? "{" Prod+ "}";
03
04  interface Prod;
05  PProd implements Prod =             Name Ext? Impl?  "=" RHS   ";";
06  EProd implements Prod = "external"  Name                       ";";
07  IProd implements Prod = "interface" Name Ext?       ("=" RHS)? ";";
08  AProd implements Prod = "abstract"  Name Ext? Impl? ("=" RHS)? ";";
09
10  Ext     = "extends"    Name ("," Name)*;
11  Impl    = "implements" Name ("," Name)*;
12
13  interface RHS;
14  Alt  implements RHS = RHS ("|" RHS)+;
15  Card implements RHS = (Group | T | NT | Enum) (opt:"?" | star:"*" | plus:"+");
16  Group  implements RHS = "(" RHS ")";
17  Concat implements RHS = RHS+;
18
19  T    implements RHS = (usage:Name ":")? "\"" Name "\"";
20  NT   implements RHS = (usage:Name ":")?      Name;
21  Enum implements RHS = (usage:Name ":")? "[" T ("|" T)+ "]";
22 }
```

Figure 3. Excerpt of the MontiCore grammar language.

(l. 2). The effect of extending a grammar is that all its productions are imported, *i.e.,* can be used in the subgrammar, and productions can be overridden. The body of a grammar comprises a list of production rules (Prod, l. 4). Production rules can be parser productions (PProd, l. 5), external productions (EProd, l. 6), interface productions (IProd, l. 7), or abstract productions (AProd, l. 8). Parser productions are grammar productions that are directly relevant for parsing. External productions underspecify a right-hand side (RHS), which must be provided by another grammar (through grammar inheritance). Therefore, a grammar containing external productions is incomplete and does not produce a parser. Abstract productions and interface productions are realized as abstract classes (or interfaces, respectively) in the abstract syntax. Parser productions and abstract productions can extend other grammar productions and override their right-hand side. The effect of this extension is that the extending grammar production can be applied wherever the extended production is expected. Interface productions can extend other interface productions. Parser productions and abstract productions may implement interface productions. The RHS of parser productions and abstract productions may contain alternatives (l. 14) and concatenations of terminals and nonterminals (l. 17) with the usual cardinalities (l. 15): optional ('?'), list ('*'), or non-empty list ('+'). Further, elements of a RHS may be contained in (nested) groups (l. 16). Nonterminals (l. 20) may have an optional name to distinguish multiple occurrences of the same nonterminal within a single RHS (e.g., to distinguish an Expression nonterminal that occurs as a precondition and as postcondition within the same RHS). Terminals (l. 19) are delimited by quotation marks and also may have an optional name. This name indicates that the terminal is relevant for the abstract syntax. A named terminal is translated to a Boolean attribute in the abstract syntax that is true iff the terminal appears in a processed model. MontiCore also supports the definition of enumeration terminals (l. 21) that translate to enumerations in the abstract syntax. The RHS of abstract productions and interface productions is optional. If no concrete RHS is present, it is underspecified and must be provided by extending or implementing productions for a grammar to be complete and produce a parser. An interface production's RHS prescribes nonterminals that have to be present in grammar productions that implement the interface productions. Furthermore, MontiCore supports literal nonterminals, which are directly translated into basic data types while they are parsed. For instance, the nonterminal Name in l. 2 defines the name of a grammar model. In the AST data structure of GrammarModel, the name is represented as a String.

3.2 Internal DSLs with Groovy

External textual DSLs [12] usually utilize a parser and employ a code generator or an interpreter to give meaning to a model. Models in internal DSLs are implemented within a host GPL, which implies that they do not require parsing. While the grammar of external DSLs enables defining any parseable concrete syntax, the freedom of design for the concrete syntax of an internal DSL is restricted to the possibilities to adapt the syntax of their host language. However, internal DSL models can be executed directly, saving the expense of a further generation and parsing process. Groovy [9, 20] is a language that is executed on the Java virtual machine and therefore integrates well with plain Java. Groovy therefore includes most of the syntax of Java, *i.e.,* many Java artifacts are valid Groovy artifacts. Compared to Java, Groovy has an optional static type system, certain simplifications of syntactical elements, and enables to specify scripts that are executed against a script base class. For instance, Groovy allows omitting the semicolon ending an instruction, omitting braces in calls of methods with at least one parameter, and omitting dots for chained method calls. Evaluating Groovy scripts requires to state a base class that defines, which properties and methods are available within the script. Groovy also supports assigning its keywords and operators new meanings. Furthermore, Groovy supports the usage of closures [9], which are anonymous blocks of statements that can have arguments and a return type. These can be used in internal DSLs for defining blocks of instructions, as they open a new scope and are delimited by curly braces.

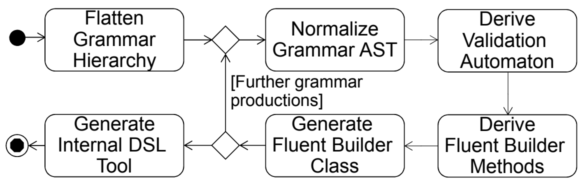

Flatten Grammar Hierarchy
Normalize Grammar AST
Derive Validation Automaton
[Further grammar productions]
Generate Internal DSL Tool
Generate Fluent Builder Class
Derive Fluent Builder Methods

Figure 4. Activities for deriving internal DSL tooling.

These particularities make Groovy well-suited for serving as host language for internal DSLs.

4 Deriving Internal DSLs

This section presents a concept for deriving a fluent builder for the abstract syntax of a language from a CFG. The fluent builder is an extension to the builder pattern [14] and enables building up the abstract syntax of a model in the fashion of an internal DSL. Deriving fluent builders can be fully automated. Besides usual CFG production rules that the parser of a language applies, our concept considers MontiCore-specific grammar production kinds and inherited grammars as presented in Section 3. The fluent builder produced by our concept yields the following properties: (1) Internal DSL and external DSL have a similar concrete syntax, (2) The internal DSL considers the order of method calls in the fluent builder to restrict the concrete syntax, (3) Fluent builders and external DSL parser instantiate the same AST data structure, and (4) The generated fluent builders are adjustable with hand-written extensions. The derivation comprises six activities as depicted in Figure 4. First, the grammar hierarchy is flattened to eliminate information not required by further steps in deriving the internal DSL. Then, the grammar is parsed, *e.g.,* with MontiCore. Afterwards, the AST of each production rule of the parsed grammar is normalized by systematically applying transformations discussed in the following. Then, we generate methods to instantiate the abstract syntax type generated from the first production of the grammar and construct a validation automaton, which validates the correct ordering of method calls. These methods are contained in a generated fluent builder class for each grammar production. If the grammar contains more production rules, the next rule is processed afterwards. Ultimately, a base class for Groovy scripts (*i.e.,* internal DSL models) defining the syntax of the internal DSL is generated. The next sections explain the activities in more detail.

4.1 Flattening Grammar Hierarchies

MontiCore allows for a grammar to inherit from multiple other grammars. For the derivation of an internal DSL, we flatten the inheritance hierarchy including all supergrammars to produce a single grammar. Overriding grammar productions might entail overriding fluent builder methods with the same name but, *e.g.,* different return types. Therefore, maintaining separate fluent builders for separate grammars that are in an inheritance relationship is not feasible. The flattening operation combines all grammar productions of all participating grammars into a single grammar. In this process, overridden grammar productions are not included in the new grammar, only the overriding productions remain. Interface nonterminals, external productions, and abstract productions are eliminated and replaced by their transitive closure. To this effect, every usage of an interface production on a RHS of a production is replaced with an alternative of all nonterminals implementing the production. For instance, the interface production `E` in Figure 1 (top) is removed and all occurrences of `E` on any RHS (in the example only in l. 2) are replaced with the alternative `(Subcomponent | Port | Connector)`. Similarly, abstract productions are eliminated. Occurrences of an abstract nonterminal on the RHS of a production are replaced with an alternative between the RHS of the abstract nonterminal and RHS of grammar productions that extend the abstract nonterminal. External productions are also removed and the transitive connection between using an external nonterminal on the RHS of another grammar production and providing a RHS to an external production is leveraged. For incomplete grammars, which contain external productions, abstract productions, or interface productions without providing grammar productions realizing their right-hand sides, no parser can be generated. Consequently, our approach does not produce fluent builders for these either. To this effect, only parser productions remain in flattened grammars. The flattening operation for grammars does, however, not alter the accepted language of the grammar. Instead, only the produced AST is altered as it does not contain any representation of interface nonterminals and abstract nonterminals. But if the realization of the concept instantiates the original AST data structure, this is feasible and internal DSL and external DSL still rely on the same AST data structure.

4.2 Normalizing the AST

As stated in Figure 3, each parser production has a `RHS` and can extend abstract productions or implement interface productions. A RHS of a parser production is a composition of seven different building blocks of two different types: composed building blocks and atomic building blocks. Composed building blocks are either `Alt`, `Card`, `Concat`, or `Group` (*cf.* Figure 3 ll. 14-17). Atomic building blocks are either `T`, `NT`, or `Enum` (*cf.* Figure 3 ll. 19-21). The result of normalizing the AST of a parsed grammar production is an intermediate data structure containing only information necessary for deriving the internal DSL. It contains information derived from the AST representing the parsed grammar and enriches it with additional information including, *e.g.,* on how to handle non-parser productions. Parsing the RHS of a grammar production produces a right-hand side AST comprising nested building blocks. Each atomic building block translates to a method in the fluent builder. Composed building blocks do not yield methods, but affect the construction of the validation automaton. This becomes more complex if non-parser productions and multiple inheritance of grammars (*cf.* Section 3.1) are considered.

The RHS of a parser production may contain terminal symbols *e.g.,* keywords, brackets, *etc.* that are not directly relevant for the abstract syntax. Such symbols usually structure the model, make it parseable, and increase its readability. The concrete syntax poses the following two challenges for deriving internal DSLs: (1) Terminal symbols typically contain special characters (*e.g.,* curly braces) or reserved keywords of the host language. For instance, `for` in imperative programming languages typically refers to a loop, while in modeling languages, it might be given a completely different meaning. This raises a problem, as keywords of the host language usually cannot be used within an internal DSL. (2) Terminal symbols are also usually used to structure the text of a model. For the internal DSL, this structure has to be "rebuilt" with means of the host language, or realized in a different way (*e.g.,* replace curly braces of a block with *begin* and *end*). To address the first issue, our approach recognizes conflicts between terminal symbols of a language that are relevant for the abstract syntax and naming guidelines for elements of the host language. These conflicts are resolved by renaming the terminal symbol. This applies to reserved keywords (*e.g.,* a terminal `final` is replaced by `r__final` to avoid name clashes with a keyword) as well as to operators (*e.g.,* a terminal + is replaced by `plus`). For terminals that are not part of the abstract syntax, the effect of their appearance in the concrete syntax for the internal DSL is not clear, therefore these are removed. This is especially necessary for deriving the validation automaton. Whenever deleting a terminal symbol, which occurs as an exclusive alternative to another building block as, for instance, in `A | B | "t"`, the whole alternative becomes optional (*e.g.,* `(A | B)?`).

The preferable realization of addressing the second issue depends on the means of the host language. In our implementation, for instance, we do not replace curly braces with *begin* and *end* statements and therefore do not perform explicit normalization for this. Instead, we leverage Groovy closures to introduce structuring elements into the concrete syntax of the internal DSL.

4.3 Deriving Validation Automata and Fluent Builders

Concatenations in the RHS of a grammar production imply a strict ordering. For instance, a parser produced for a grammar production `C = x y` always expects to parse an `x` that is followed by a `y`. If this is not the case, the parser rejects the model. Method calls of builders [14], on the other hand, are generally not restricted in the order or cardinality of invocations. This raises a gap when using builders for an internal DSL. For example, the method call trace `y() y() x()` in the internal DSL would be allowed whereas the corresponding grammar only accepts the trace `x() y()`. Furthermore, a builder usually would not distinguish between alternatives and concatenations. To avoid accepting invalid traces, our approach produces a validation automaton for each production in the grammar. The automaton checks correct method ordering at runtime and is contained in the generated fluent builder class.

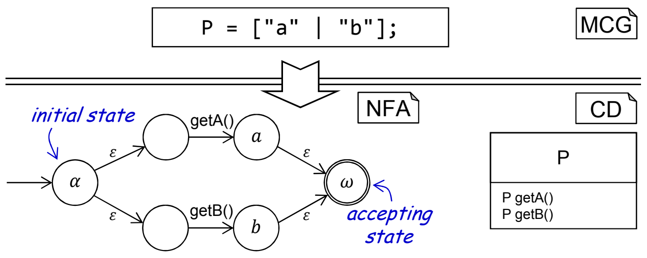


Figure 5. Derivation of a validation automaton and fluent builder class from an exemplary constant group.

Whenever a fluent builder method is called, the automaton tries to accomplish the corresponding transition that starts at the current internal state and matches the transition label. If there is no such transition, an appropriate error is thrown. The validation automaton can be intuitively interpreted as the automaton corresponding to a regular expression matching the RHS of the parser production constructed by applying the Thompson construction [29]. Based on a set of rules, parts of the validation automaton are produced. Then, these automata are connected to realize alternatives and concatenation. The following explains the derivation rules for creating the validation automaton and corresponding methods generated from the RHS of grammar productions.

Rule 1: Terminal Symbols Relevant for Abstract Syntax As introduced in Section 3, the MontiCore grammar language supports terminal symbols that reflect in the abstract syntax, either as named terminal symbol or as constant group (*i.e.,* enumeration). Constant groups are more general than named terminals, as the latter can be transformed into a constant group comprising only a single constant. Therefore, we only consider a derivation rule for constant groups.

Figure 3 depicts the `Enum` production (l. 21) that defines constant groups in the MontiCore grammar language. Constant groups are atomic building blocks and as such, affect the fluent builder as well as the validation automaton. For each constant of a constant group in a parser production `P`, we generate an attribute and a getter method for the property into the corresponding fluent builder class. This is depicted in Figure 5 (right) by example. The produced validation automaton (part) has a single source and a single accepting state. Each constant of a constant group is translated into two states and a transition labeled with the getter method of the generated property in the fluent builder. The source state

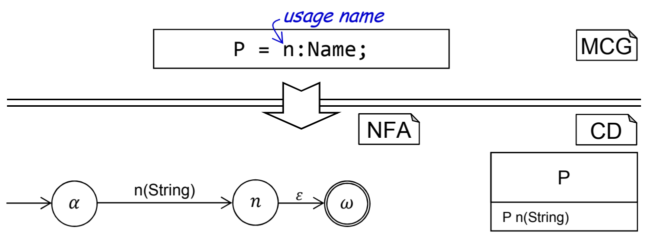


Figure 6. Derived validation automaton and fluent builder class from production `P = n:Name`.

of the automaton is connected to the first state of each alternative, and the second state of each alternative is connected to the joint accepting state. For each constant, this automaton therefore contains exactly one path from source state to accepting state. This is depicted by example in Figure 5 (left), where one path implies to invoke the method `getA()` generated from the constant `a` and the other path implies to invoke the method `getB()` generated from the constant `b`.

Rule 2: Nonterminal Usage The RHS of grammar productions may contain nonterminal symbols that refer to the nonterminal of a different grammar production, therefore, they "use" the nonterminal. Referencing a nonterminal has no visible impact on the concrete syntax of a language, because all occurrences of nonterminal references can be replaced by their corresponding referenced production's RHS until a single rule is left. However, this approach results in an exponential blowup of this rule and the transformation may not terminate if the input grammar contains cyclic dependencies between rules. Thus, such an in-lining of nonterminals is not possible in general.

Instead, our approach generates one fluent builder method per nonterminal usage. For the name of the method, we leverage the usage name of the nonterminal (*cf.* Figure 3) to distinguish potential usages of the same nonterminal in the same RHS of a production. If no such usage name is present, it is derived from the nonterminal name. The derived nonterminal usage name is equal to the nonterminal name, starting with a lower case letter. The generated fluent API method always has a single parameter, which is a closure switching to the namespace of the used nonterminal. Each nonterminal (usage) name is transformed to a name compliant with the naming conventions of methods in the host language of the internal DSL. As depicted in Figure 6, constructing a corresponding validation automaton for a nonterminal reference is straightforward. It comprises three states: an initial state α, a method call state n, and a final state ω. A transition is triggered if the corresponding method derived from nonterminal usage is executed. Figure 6 visualizes the fluent builder class `P` generated for production `P = n:Name` and the corresponding validation automaton contained in `P`. The method has a parameter of type `String` (and not a closure), because `Name` is a special nonterminal – a literal – that is translated to a String. Therefore, no closure is required.

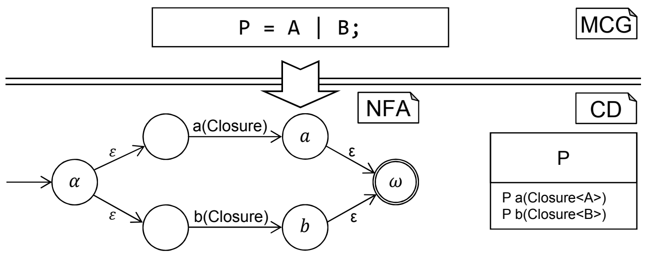


Figure 7. Derived validation automaton and fluent builder class from production `P = A | B`.

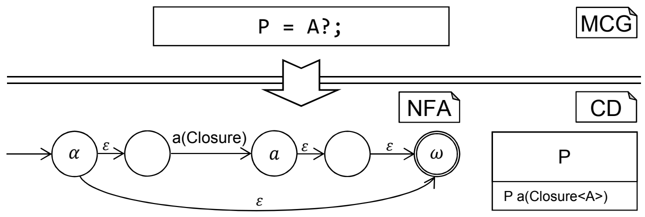


Figure 8. Derived validation automaton and fluent builder class from production `P = A?`.

Rule 3: Alternatives The syntax of alternatives is depicted in Figure 3 (l. 14). Alternatives are composed building blocks, and as such for these no fluent builder methods are generated. Nonetheless, they affect the construction of the validation automaton. An alternative node has at least two child nodes, one for each option. For each option, we derive a separate validation automaton. The Thompson construction for all alternatives results in a validation automaton with initial state α and accepting state ω. Similar to the derivation of the validation automaton for constant groups, different paths – one per option of the alternative – lead from the initial state to the accepting state. Transitions are labeled with the methods generated from the nonterminal usages, if an alternative directly contains a nonterminal usage. Otherwise, the application of the Thomson construction is continued and the alternative paths contain other parts of the overall validation automaton derived from the respective alternatives in the grammar. A concrete example for production `P = A | B` is given in Figure 7. The fluent builder class `P` contains methods generated from the nonterminal usages of `A` and `B`. The corresponding validation automaton yields two paths from initial state to accepting state.

Rule 4: Cardinalities In MontiCore, cardinalities are: optionals (?), iterations (*), or non-empty iterations (+), as introduced in Figure 3 l. 15. All cardinalities are composed building blocks and therefore only affect the construction of the validation automaton. Each of the three cardinalities yields a separate derivation rule. The generated validation

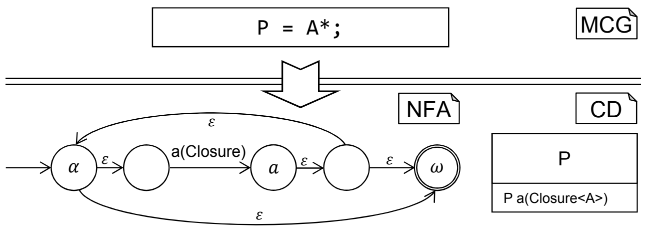


Figure 9. Derived validation automaton and fluent builder class from production `P = A*`.

automaton for an optional contains the initial state α, the accepting state ω state, and three further states. Two paths connect the initial state with the accepting state. One path includes all states from initial state to final state and contains a transition labeled with the method call for the nonterminal usage. The other path, representing that the option has not been chosen, leads directly from initial state to target state without the transition labeled with the method call. Figure 8 depicts the fluent builder class and the validation automaton for the exemplary production `P = A?`.

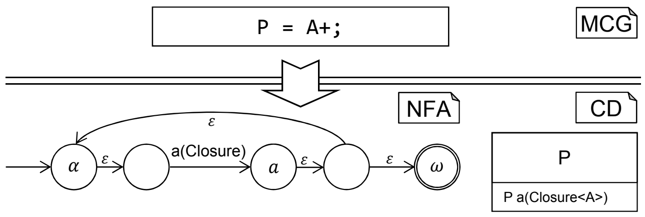


Figure 10. Derived validation automaton and fluent builder class from production `P = A+`.

For iterations, the constructed validation automaton, depicted in Figure 9, is similar to the one generated for optionals. Additionally, it contains an edge connecting the state after the target state of the transition labeled with the method call to the initial state. This edge realizes the iteration.

For other non-empty iterations, the validation automaton is similar to the one for iterations, but the edge from initial state to accepting state is removed. As effect, the edge labeled with the method call has to be visited at least once. An example for production `P = A+` is depicted in Figure 10 that includes the generated fluent builder class and the resulting validation automaton.

Rule 5: Concatenation and Groups Concatenation is a composed building block (*cf.* Figure 3, l. 16) describing a sequence of other building blocks. Again, as concatenation is a composed building block, we do not generate dedicated fluent builder methods. The derived validation automaton concatenates the automata of all child nodes in the correct order. The initial state of the resulting automaton is the initial state of the first automaton in the concatenation. The

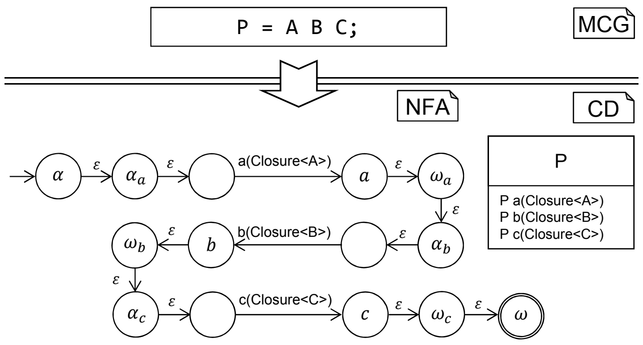


Figure 11. Derived validation automaton and fluent builder class from production `P = A B C`.

accepting state of the concatenation is the accepting state of the last automaton in the concatenation. The intermediate automata are joined by connecting the accepting state of the i-th element of the concatenation with the initial state of the $i + 1$-th element of the concatenation. This is depicted by example in Figure 11.

In some cases, it is necessary to group certain building blocks. For instance, the concrete syntax of `(AB)*` in most cases differs from the concrete syntax of `A*B*` and the semantics might as well. The latter is due to the fact, that the numbers of `A` appearances and `B` appearances must be equal in `(AB)*`, whereas it may differ in `A*B*`. In MontiCore grammars, groups are composed building blocks enclosed by parentheses that contain a non-empty set of sub-blocks. In the grammar depicted in Figure 3, this is indicated by the fact that a group may contain any `RHS`. Group nodes can affect the construction of validation automata, *e.g.*, `a|b*` results in a different validation automaton than `(a|b)*`. The second case allows a trace `bb` which is not supported by the first case. But the validation automaton for `(a|b)` does not differ from the automaton for `a|b`. Groups only affect the order in which parts of the validation automaton are derived.

5 Deriving Internal Groovy DSLs from MontiCore Grammars

The internal DSL is designed as alternative to the external DSL parser, where both instantiate the same AST data structure to leverage reusing tooling and code generators for both. The derivation of the internal DSL is integrated into the existing MontiCore infrastructure by generating an external DSL parser and AST data structure, as visualized in Figure 12. The fluent builder is generated by the fluent builder generator that utilizes the MontiCore grammar parser for processing the input grammar. The generated fluent builders can be optionally customized by extending these with handwritten Java classes (*cf.* Section 5.3). The fluent builder is executed by the internal DSL tooling that processes an internal DSL model, *i.e.*, a Groovy script.

5.1 Generating Fluent Builder Classes

The internal DSL is a collection of fluent builder classes implemented in Java. Each fluent builder class is a fluent builder for a MontiCore AST node. Each method is generated from one leaf node of the normalized AST and employs the generated validation automaton to check the validity of a method invocation. Generating a fluent builder class does not depend on other fluent builder classes, therefore the generation can be parallelized. The validation automaton is realized as a state map mapping transition labels to a set of possible valid predecessor states. This yields an efficient check for deciding at the beginning of each method whether calling the method is currently allowed or not. If this check evaluates to true, the current state is updated properly. Otherwise, an error is thrown. The predecessor state sets for all transition labels are computed while generating the fluent builder class. First of all, the validation automaton produced from the grammar according to the rules presented in Section 4 has to be transformed into an equivalent deterministic final automaton without ε transitions by applying powerset construction [26]. After that, the predecessor state set for each transition label is computed.

A fluent builder contains literal methods, closure methods, or Groovy properties, depending on the productions contained in the input grammar. All three are explained in the following: In MontiCore, each grammar may define special literal productions that translate to basic data types. These include, *e.g.,* a `BooleanLiteral` that translates to Boolean, an `IntLiteral` that translates to an integer, or `Name` that translates to a string. If a nonterminal reference on the RHS of a production refers to such a literal production, MontiCore translates these into attributes of the corresponding AST class. The fluent builder generator creates one method for each literal nonterminal reference. The method header has the form `A name(T value)` where `A` is the name of the fluent builder class and `T` is the basic data type that the literal translates to. `name` is either the usage name or the name of the referenced production if the usage name is not present (*cf.* Figure 3, l. 20). Whenever a literal method is executed, it validates the current state of the validation automaton and instantiates the respective AST class.

A closure method has the form `A name(Closure<B> cl)` where `A` is, again, the name of the current fluent builder class and `B` is the name of a fluent builder class produced from a referenced (non-literal) grammar production. Calling a closure method validates the current builder state and – if valid – applies one recursion step. Therefore, the method creates a new fluent builder class instance of the builder referenced in the closure and executes this closure. The executed code within the `cl` utilizes the new fluent builder class instance and produces a correctly populated AST node of the referred production, which is then set as a child node of the current AST node, *i.e.,* the node instantiated by the current builder.

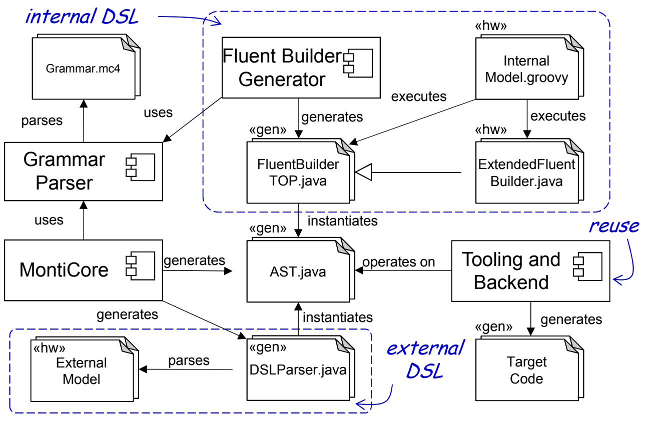


Figure 12. Integration of the internal DSL infrastructure into the MontiCore language workbench.

The third kind of methods are Groovy properties, which are of the form `A getInitial()` where `A` is the name of a fluent builder class and `initial` is the name of a property. They are generated for each constant in a constant group and for each terminal that is visible within the AST (*cf.* Section 3.1). Groovy properties are similar to literal methods, *i.e.,* they validate the current builder state and set the appropriate field of the current AST node.

5.2 Using the Fluent Builder in Groovy

For better compatibility with existing MontiCore infrastructure, the fluent builder classes are generated as Java classes (and not as Groovy classes). Therefore, the internal DSL model implemented as Groovy script has to be aware of the available methods and properties of the fluent builder. To this effect, a special `BaseScript` class is generated that contains all possible fluent builder start methods. These start methods are those derived from start productions of the grammars, *i.e.,* those that produce a root node of the AST. If the start production is a parser production, there is exactly one method. In case the start production is an abstract or interface production there may be multiple start methods. The `BaseScript` class is set as base class to the internal DSL model that is to be processed. Furthermore, a so-called internal DSL "parser" is generated. Analogously to an external DSL parser, it accepts an internal DSL model and produces a corresponding AST of the same type. Therefore, all MontiCore infrastructure generated for an external DSL can be reused without any modification. In other words, parsing an external DSL model and the corresponding internal DSL model produces equal ASTs.

5.3 Customizing Derived Fluent Builder

The syntax of automatically generated internal DSLs may not conform to the language developer's intent and may not be intuitive to understand. The question, if the presented

```
01 public P2 p1(String name) {                                Java
02   return p1(closure(p1 -> p1.name(name)));
03 }
```

Figure 13. Example of preventing a closure.

```
01 public P1Chain p1(String name) {                           Java
02   return new P1Chain(name);
03 }                                                 return type
06 public class P1Chain extends Chain<P1, P2> {
07   public P1Chain(String name) { … }
08   public P2 age(int age) {  type of serving fluent builder
09     getBuilder().age(age);
10     return end();
11   }                          delegates to P1
12 }            ends the chain
```

Figure 14. Example of method chaining.

approach produces "good" results mainly depends on the shape of the given input grammar and the set of implemented derivation rules. Especially, this approach derives method names from corresponding nonterminal names, which can lead to unexpected naming in the fluent builder. To overcome this, our concept supports handwritten extensions of the generated fluent builder.

To extend a fluent builder class with a handwritten class, we apply the MontiCore TOP mechanism [28] for AST nodes to the fluent builder. During generation, the fluent builder generator checks if there exists a (handwritten) class in the expected package with the same name. If so, the generator adds the TOP postfix to the generated fluent builder class name. The handwritten class has to extend the corresponding TOP class and therefore inherits all generated fluent builder methods. To this extent, the TOP mechanism only allows addition and overriding of existing methods but does not support to delete generated methods. Furthermore, the implementation of a fluent builder method can use other fluent builder methods, but must not access the AST directly. In other words, all added handwritten methods may only use the generated fluent builder and are, therefore, shortcuts for inconvenient parts of the internal DSL.

On the one hand, this approach has the drawback that it is limited in its expressiveness, especially if the language developer intents to explicitly remove undesired fluent builder methods. On the other hand, it has the advantage that the fluent builder is backward compatible to the generated version. Additionally, it is not necessary to apply any modifications to the validation automaton, because reusing other fluent builder methods still guarantees that the constructed AST node is built correctly. We have identified four types of common fluent builder modifications sufficient to adapt the concrete syntax of an internal DSL towards the concrete syntax of the corresponding external DSL:

```
01 public P2 p1(String name, int age) {                       Java
02   return p1(closure(p1 -> p1.name(name).age(age)));
03 }
```

Figure 15. Example of multi-parameter methods.

Closure prevention: Sometimes, the RHS of a production is trivial *e.g.,* `P1 = Name`. In this case, the generator introduces a closure method for each reference of `P1` which results in an inconvenient model *e.g.,* `p1 { name "user" }`. By adding a closure preventing method as shown Figure 13 into the fluent builder class `P2`, the language developer avoids this flaw. The closure method used in l. 2 adapts the given lambda expression as a Groovy closure and passes it to the fluent builder method `p1`.

Method chaining: Method chaining is another concept of closure prevention, which can be applied to replace an arbitrary closure method. Method chaining replaces a closure by a concatenation of method calls. Based on the validation automaton, this replacement could be derived fully automatically. However, this would yield the drawback that the concrete syntax of the internal DSL may be more complicated to understand. As depicted in Figure 14, method chaining for a production `P1 = Name age:IntLiteral` can be realized by adding an additional nested class `P1Chain` to fluent builder class `P2`. This `P1Chain` creates a new internal fluent builder of `P1`. All methods of `P1Chain` delegate to corresponding methods of `P1` (*e.g.,* in l. 9). If there is no following method in the chain, `P1Chain` invokes the fluent builder method `p1(Closure)` in `P2` with the internal fluent builder object (`P1`) as argument. In this example, method chaining allows the modeler to write `name "user" age 42` instead of `p1 { name "user" age 42 }`.

Multiple parameter methods: By default, generated fluent builder methods always have a single parameter. This can be changed by adding a handwritten method with multiple parameters. In the internal DSL model, multiple parameters must be separated by a comma. For example, Figure 15 defines a method supporting the concrete syntax `p1 "user", 42`. The implementation is similar to Figure 13.

Adding operators and renaming methods: Besides closure prevention, method chaining, and multi-parameter methods, there are other possibilities to modify the generated fluent builder. For example, the language developer can create additional methods behaving as aliases for other fluent builder methods or add operators to the syntax of the internal DSL. The latter leverages the possibility of Groovy to overload operators (*e.g.,* a bitwise right shift operator) and to give these a new meaning. Furthermore, all mentioned modifications can be combined with each other.

```
01 grammar Automaton extends Literals {                                 MCG
02   Automaton = "automaton" Name "{" (State | Trans)+ "}";
03   State = "state" Name (("<<" ["initial"] ">>") |
04            ("<<" ["final"] ">>"))*
05            (("{" (State | Trans)* "}") | ";");
06   Trans = from:Name "-" input:Name ">" to:Name ";";
07 }
```

Figure 16. Initial automaton grammar. Parts affecting the syntax of the generated fluent builder are highlighted blue.

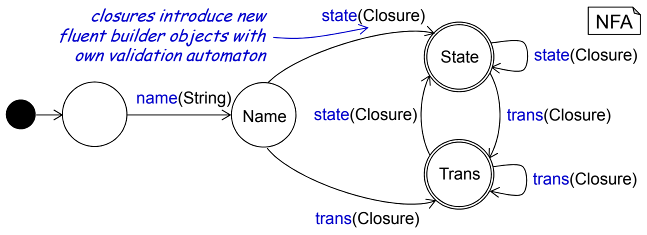


Figure 17. Validation automaton for the rule Automaton.

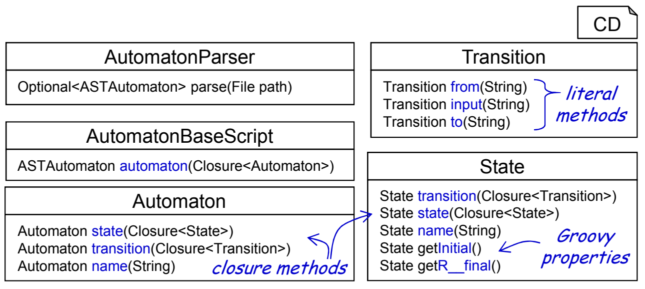


Figure 18. Generated fluent builder classes.

```
automaton TrafficLight {                                   Automaton
  state Red <<initial>>;
  state Yellow;
  state Green <<final>>;

  Red - next > Yellow;
  Yellow - next > Green;
}
```

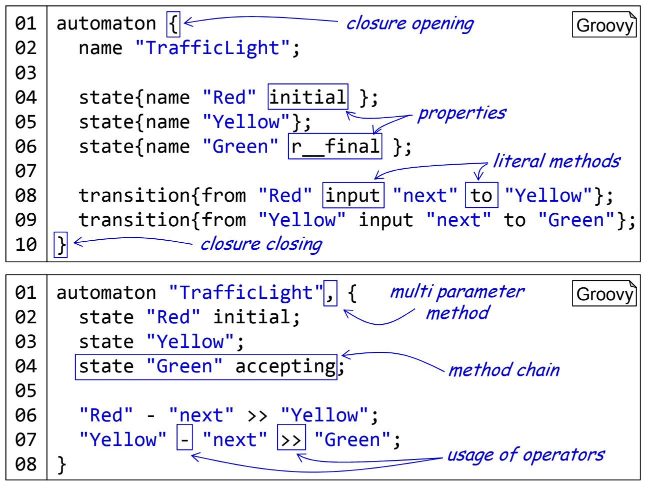
Figure 19. Example of an external DSL model.

6 Case Study

Engineering a modeling language requires the profound design decision whether to develop an internal DSL or an external DSL. As this affects how language tooling is developed, this decision is usually taken at an early stage in development to avoid re-engineering of tooling. With our approach, switching between both kinds of DSLs causes much less effort in re-engineering. Our implementation generates the complete infrastructure for the internal DSL. Optionally, the automatically derived internal DSL can be customized with handwritten enhancements, which are realized by subclassing the generated artifacts. Consequently, the generated code is not directly manipulated, but extended with handwritten customizations. For illustration, we consider the development of an automaton language. The automaton language can be used for modeling discrete state transition systems. It supports the definition of (hierarchically decomposed) states and transitions. Hierarchical states include a body comprising further states and transitions. The grammar of the automaton language is developed by the language developer and is depicted in Figure 16. It can be used as a starting point for deriving the parser of the external DSL as well as the generation of the fluent builder for the internal DSL. The automaton grammar includes parser productions (ll. 2,4,7), terminals (ll. 2-6), literal (ll. 2,3,6) and non-literal (ll. 2,5) nonterminal references, constant groups (ll. 3-4), together with composed building blocks iteration, disjunction, grouping, and concatenation.

To preserve the ordering of the right-hand sides, the fluent builder generator constructs a validation automaton for each production. Figure 17 depicts a simplified version of the generated automaton for production Automaton. For each fluent builder method, there exists a related state in the automaton. All incoming transitions of a state are labeled with the corresponding method name. The states State and Trans are marked as final. This fits the observation that the Automaton production requires the definition of a name and at least one state or transition, whereas further states or transitions are optional. As depicted in Figure 18, the fluent builder generator produces one fluent builder class per production and the two classes AutomatonBaseScript and AutomatonParser. Non-literal nonterminal references are mapped to methods with a closure as parameter as, *e.g.*, state(Closure). Because the automaton grammar only uses the Name literal production, all literal methods accept a String. Furthermore, Groovy properties and get-methods for these are generated for each constant of a constant group *e.g.*, getInitial(). Counter-intuitively, methods with the prefix get set an AST attribute to a certain value of a property. Since Groovy allows omitting get-prefixes together with following brackets, this does not influence the concrete syntax of the internal DSL. A profound drawback of the generated fluent builder is the naming of the method r__final. As final is a keyword of the host language, it can neither be used as a method nor as property name. To overcome this, our generator adds the prefix r__, which makes the fluent builder and therefore the internal DSL unintuitive.

After executing the fluent builder generator, the internal DSL prototype is immediately available to the customer for early feedback. The top of Figure 20 depicts an example traffic light model, written in Groovy and utilizing the generated fluent builder. At the beginning of each internal DSL model, the developer can only use fluent builder methods provided in the AutomatonBaseScript class *i.e.*, in the example, only



```
01 automaton {
02   name "TrafficLight";
03
04   state{name "Red" initial };
05   state{name "Yellow"};
06   state{name "Green" r__final };
07
08   transition{from "Red" input "next" to "Yellow"};
09   transition{from "Yellow" input "next" to "Green"};
10 }
```

```
01 automaton "TrafficLight", {
02   state "Red" initial;
03   state "Yellow";
04   state "Green" accepting;
05
06   "Red" - "next" >> "Yellow";
07   "Yellow" - "next" >> "Green";
08 }
```

Figure 20. Example of an internal DSL model using the generated (top) and extended fluent builder (bottom).

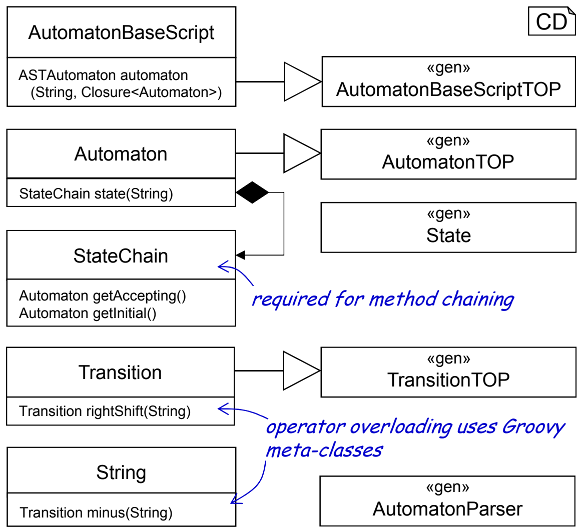


Figure 21. Class diagram of the extended fluent builder.

the method automaton(Closure<Automaton>). We can observe the effect of closures on the example model, which change the available methods. All fluent builder methods that require a parameter Closure<T> also introduce a pair of curly brackets into the model. Inside these brackets, the modeler can only choose methods provided by fluent builder class T. Definitions of state "Red" (l. 4) and "Green" (l. 6) invoke the property methods getInitial() and getR__final().

Since the effort of generating an appropriate external DSL parser is negligible and possible at any point in time, the customer may take a look at external DSL models resulting in the same abstract syntax as the internal DSL model, *e.g.*, as depicted in Figure 19. Here, a list of potentially undesired differences between both models can be detected: The definitions of states and transitions in the internal DSL introduce curly brackets that are not present in the external DSL. The name method inside the definitions of automaton and state also change the appearance of the language. Moreover, the r__final is unintuitive and could be replaced by a keyword that is not reserved in the host language, *e.g.*, accepting.

These enhancements can easily be implemented manually using the fluent builder TOP mechanism as described in Section 5.3. In our example, the developer has applied some enhancements of the internal DSL's concrete syntax by providing handwritten classes that extend the generated classes of the internal DSL as depicted in Figure 21. Classes marked with the «gen» stereotype are equal to the classes presented in Figure 18, the postfix TOP indicates that MontiCore found handwritten classes acting as extension to the generated ones. Unmarked classes are handwritten.

The developer adds a method state(String) enabling to chain consecutive calls of getInitial() or getAccepting(). Therefore, the class Automaton extends the appropriate TOP-class and adds the method state(String) returning an instance of StateChain, which enables calling getInitial() or getAccepting(). Consequently, the curly brackets and the name keyword can be omitted and r__final is replaced by accepting. Also, the developer adds a fluent builder method with two parameters to class AutomatonBaseScript. Thus, the name of the automaton can directly be defined without any additional call of name. As depicted in Figure 20 l. 1, arguments of a multi-parameter method are separated by comma. Moreover, operator overloading via Groovy meta-classes [20] enables replacing from, input, and to by the operators - and >>, resulting in a concrete syntax closer to the external DSL. As overloading > is not supported, the language developer chooses >> instead. Whether the presented adjustments to the concrete syntax are "improvements" depends on the taste of the internal DSL developer or customer. They can be combined, but fluent builder methods can only be added or modified, never removed. Whenever the customer decides to prefer the internal over the external DSL or vice versa, developed language tooling or infrastructure do not have to be changed. This makes language engineering more agile and enables "rapid prototyping" of DSLs.

7 Discussion

We derive internal DSLs from grammars by generating fluent builders in Groovy that can operate on the grammars' external abstract syntax representations. This enables using both shapes of the DSL - external and internal - in parallel and facilitates developing language tooling. However, giving meaning to a processed internal DSL model requires to interpret the respective AST or to generate executable (GPL) code from it. Being fully automated also facilitates engineering internal DSLs with our approach, as developing these requires complex host language patterns (such as closures) that introduce accidental complexities [13] to language engineering rarely unbeknownst to grammarware. Their syntax of internal DSLs is naturally restricted by their host language. Many restrictions can be mitigated using the presented extension mechanism, some restrictions, *e.g.,* reserved language keywords, cannot be lifted easily. While selecting another host language may enable using the internal DSL keywords of choice, our concept rests on the existence of closures or a similar concept. Its translation into other host languages, such as Scala [24], is subject to investigation. Our concept, furthermore, relies on flattening the inheritance hierarchy of languages, which (a) generally can lead to name clashes between inherited productions and thus render the resulting grammar unusable, and (b) is inefficient as certain base grammars must be translated more often than necessary instead of reusing their internal DSL representations between different language projects. Regarding the former, we assume that the languages' grammars are valid with respect to inheritance, *i.e.,* either their names can be qualified unambiguously or the originating language workbench already takes care of rejecting such grammars. We currently cannot prevent re-generating the fluent builders. The naive approach of inheriting from builders generated from super-grammars previously fails on the type systems of relevant host languages. For instance, we cannot have methods of the same name and different return types (*i.e.,* abstract syntax types). This also is future work.

8 Related Work

There are various language workbenches [11] that support creating external or internal DSLs.

MPS [25] relies on projectional editing for (meta)model modification. Its base language, a DSL similar to Java, can be extended to support stepwise derivation of an internal DSLs by extending the base language with internal DSL concepts. However, without significant manual adjustments, both shapes share the abstract syntax of Java, which can be more complex than necessary. Neverlang [31], Rascal [19], Spoofax [33], and Xtext [3] support defining external DSLs as grammars (Neverlang, Rascal, Xtext) or abstract data types (Spoofax) and focus on different aspects of engineering external DSLs. Neither deriving internal DSLs, nor integrating the abstract syntaxes of both shapes are supported. Regardless, developing a generator that translates grammars into a fluent API as presented in this paper would be realizable in these language workbenches. Another approach [7] translates UML class diagrams into an internal Java DSL. The implementation processes XML models and translates these into model graphs. These are translated into Java Fluent APIs using a template engine. While similar, this approach is limited to class diagrams. MetaBorg [6] enables providing a customized concrete syntax in a GPL. To this end, the metaprogrammer defines a new concrete syntax and an assimilation that describes how code specified using the new syntax translates into code of the host language. The paper proposes an implementation with SDF [33] for syntax definition and SGLR [5] as a parser. This requires a syntax definition of the host language which, in general, is not present and has to be defined first. Racket [30] is a programming language supporting the definition of internal DSLs. These can be bundled into modules to be reused in different projects. The framework generally supports the definition of DSLs but does not allow to change between internal and external DSLs. Cedalion [23] realizes the inverse of our concept: Its projectional editor facilitates the definition of an individual concrete syntax that is independent on the host language. Thus, it supports creating internal DSLs and providing these with external DSL features as an individual concrete syntax. With this at hand, developers can offer DSL-specific tool support, *e.g.,* editors or analyses for internal DSLs. Xbase [10] is a DSL that provides Java expressions and statements enriched with some syntactic sugar. Developers can combine Xbase with any Xtext DSL to enrich their models with behavior descriptions. These can be interpreted or generated into Java code via the provided generator. Hence, Xbase is an external DSL providing Java features while we translate external DSLs definitions (grammars) into internal DSLs. Overall, to the best of our knowledge, there are currently no other concepts or language workbenches supporting the integrated engineering of external DSLs and internal DSLs such that these operate on the same data structures.

9 Conclusion

We have presented a concept to derive internal DSLs from grammars that enables agile language prototyping before committing to an internal or external DSL shape. To this end, we automatically derive fluent builders from the grammars' productions that instantiate classes of the grammars' external abstract syntax classes. Instantiating models of the internal DSL is as restricted as for models of the external DSL due to the accepting automata added for each grammar rule. These prevent constructing invalid models with respect to the grammar. We plan on conducting a study evaluating the effort of creating an internal DSL from scratch compared to creating an internal DSL with our approach. Further, we are currently developing a concept for executing abstract syntax classes that, in combination with the presented approach, fosters agile language engineering. Overall, the presented approach facilitates agile language prototyping, as tooling (analyses, well-formedness rules, code generators) can be developed against abstract syntax classes while exploring the different shapes with customers.

References

[1] Mathieu Acher, Benoit Combemale, and Philippe Collet. 2014. Metamorphic domain-specific languages: A journey into the shapes of a language. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 243–253.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

[3] Konstantinos Barmpis, Dimitrios Kolovos, and Justin Hingorani. 2018. Towards a Framework for Writing Executable Natural Language Rules. In *European Conference on Modelling Foundations and Applications*. Springer, 251–263.

[4] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. 2016. Execution Framework of the GEMOC Studio (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, ACM, New York, NY, USA, 84–89.

[5] Martin Bravenboer, Eelco Dolstra, and Eelco Visser. 2007. Preventing Injection Attacks with Syntax Embeddings. In *Proc. of International Conference on Generative Programming and Component Engineering (GPCE) 2007*. ACM. https://doi.org/10.1145/1289971.1289975

[6] Martin Bravenboer and Eelco Visser. 2004. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 365–383.

[7] Dmitry Buzdin and Oksana Nikiforova. 2012. Transformation of UML class diagram to internal java domain-specific language. *Applied Computer Systems* 13, 1 (2012), 61–67.

[8] Jesús Sánchez Cuadrado, Javier Luis Cánovas Izquierdo, and Jesús García Molina. 2013. Comparison between internal and external DSLs via RubyTL and Gra2MoL. In *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. IGI Global, 109–131.

[9] Fergal Dearle. 2010. *Groovy for Domain-Specific Languages*. Packt Publishing Ltd.

[10] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. 2012. Xbase: implementing domain-specific languages for Java. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 112–121.

[11] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. 2015. Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. *Computer Languages, Systems & Structures* 44 (2015), 24–47.

[12] Martin Fowler. 2010. *Domain-Specific Languages*. Addison-Wesley Professional.

[13] Robert France and Bernhard Rumpe. 2007. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE*.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[15] Object Management Group. 2010. Object Constraint Language Version 2.2 (OMG Standard 2010-02-01).

[16] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Voelkel, and Andreas Wortmann. 2015. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*. Scitepress, Angers, France.

[17] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. 2015. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*. ACM/IEEE, 136–145.

[18] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. 2015. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling* 14, 2 (2015), 905–920.

[19] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. 2009. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*. IEEE, 168–177.

[20] Dierk König, Paul King, Guillaume Laforge, Hamlet D'Arcy, Cédric Champeau, Erik Pragt, and Jon Skeet. 2015. *Groovy in Action*. Manning Publications.

[21] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2008. MontiCore: Modular Development of Textual Domain Specific Languages. In *Proceedings of Tools Europe*.

[22] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2010. MontiCore: a Framework for Compositional Development of Domain Specific Languages. In *International Journal on Software Tools for Technology Transfer (STTT)*.

[23] David H Lorenz and Boaz Rosenan. 2011. Cedalion: a language for language oriented programming. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 733–752.

[24] Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in scala*. Artima Inc.

[25] Vaclav Pech, Alex Shatalin, and Markus Voelter. 2013. JetBrains MPS as a tool for extending Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. ACM, 165–168.

[26] Michael O Rabin and Dana Scott. 1959. Finite automata and their decision problems. *IBM journal of research and development* 3, 2 (1959), 114–125.

[27] Alastair Reid, John Peterson, Greg Hager, and Paul Hudak. 1999. Prototyping real-time vision systems: An experiment in DSL design. In *Proceedings of the 21st international conference on Software engineering*. ACM, 484–493.

[28] Bernhard Rumpe and Katrin Hölldobler. 2017. *MontiCore 5 Language Workbench. Edition 2017*. Shaker Verlag.

[29] Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (June 1968), 419–422.

[30] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as libraries. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 132–141.

[31] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures* 43 (2015), 1–40.

[32] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. 2014. Sirius: A Rapid Development of DSM Graphical Editor. In *Intelligent Engineering Systems (INES), 2014 18th International Conference on*. IEEE, IEEE, Tihany, Hungary, 233–238.

[33] Guido H Wachsmuth, Gabriël D P Konat, and Eelco Visser. 2014. Language Design with the Spoofax Language Workbench. *IEEE Software* 31, 5 (2014), 35–43.