A Taxonomy of Change Types for Textual DSL Grammars

Hossain Muhammad Muctadir[1][a], Jérôme Pfeiffer[2][b], Judith Houdijk[1], Loek Cleophas[1][c]
and Andreas Wortmann[2][d]

[1]*Software Engineering and Technology cluster, Eindhoven University of Technology, Eindhoven, The Netherlands*
[2]*Institute for Control Engineering of Machine Tools and Manufacturing Units, University of Stuttgart, Stuttgart, Germany*

Keywords: Change Taxonomy, Domain Specific Language, Textual DSL Grammar.

Abstract: Domain-Specific languages (DSLs) bridge the gap between the domain-specific problem space and the solution space of software engineering. Engineering DSLs is a complex and time-intensive iterative process involving exchanges with stakeholders who amongst others decide on the DSL's syntax. Since in this process the stakeholder requirements change frequently, so can the corresponding DSL. The subsequent changes to the language specification may produce conflicts that language engineers need to be aware of and resolve. Current research has not adequately answered the question which change operations for grammar-based syntax exist, and which impact they have at meta-model and model level. To answer this question we develop a taxonomy of change types for grammars of textual DSLs that includes the concepts typically found in grammar-based language workbenches such as Xtext, MontiCore, and Neverlang, and lists the possible change operations that can be performed. The taxonomy was built iteratively based on an Xtext based implementation of the Systems Modeling Language v2 and evaluated in a case study that leverages the taxonomy to perform impact analysis. The taxonomy presented in this paper will help language engineers to analyse the impact of changes to the grammar-based syntax specification of a language and to utilize this analysis, e.g., to perform historical change impact analysis.

1 INTRODUCTION

In software system development, domain-specific problems are often not easily represented with general-purpose programming languages (GPLs). Domain specific languages (DSLs) (Hölldobler et al., 2017) in contrast allow for domain-specific description of the intended solution (Hölldobler et al., 2019). While most DSLs are initially developed as small languages with limited syntax and capabilities (Butting et al., 2020), they are typically extended to meet the increasing demand as the DSLs are used for real world problem solving (Thanhofer-Pilisch et al., 2017; Mengerink et al., 2016; Zhang and Strüber, 2024). This evolution of DSLs often needs to be propagated to various related artifacts, which is known to be error prone (Meyers and Vangheluwe, 2011). We focus on taxonomizing possible change types for DSL grammar changes, as a first step for developing a structured method for analysing the impact of and propagating such changes.

Nowadays, the development and maintenance of DSLs are accomplished using various language workbenches (LWB). Most modern LWBs focus heavily on reuse, allowing importing, inheritance, and extension of existing grammars and meta-models. We aim to study the impact of changes made to textual grammar definitions on other dependent grammars. For this purpose, we developed a taxonomy of changes for textual DSL grammars and utilised it in a case-study, where we performed change impact analysis (CIA) on Xtext-based[1] grammar definitions of the pilot implementation of the second version of SysML, the Systems Modeling Language (SysML-V2) (Seidewitz et al., 2023).

This paper is structured as follows. Section 2 covers related work on DSL evolution. In Section 3, we detail our method for developing the taxonomy and present it. Section 4 discusses the Xtext-based case-

[a] https://orcid.org/0000-0002-2090-4766
[b] https://orcid.org/0000-0002-8953-1064
[c] https://orcid.org/0000-0002-7221-3676
[d] https://orcid.org/0000-0003-3534-253X

[1] Xtext https://eclipse.dev/Xtext/

Muctadir, H. M., Pfeiffer, J., Houdijk, J., Cleophas, L. and Wortmann, A.
A Taxonomy of Change Types for Textual DSL Grammars.
DOI: 10.5220/0013127800003896
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 13th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2025)*, pages 169-176
ISBN: 978-989-758-729-0; ISSN: 2184-4348
Proceedings Copyright © 2025 by SCITEPRESS – Science and Technology Publications, Lda.

study and the applicability of the taxonomy to other LWBs. Finally, Section 5 explores future works and concludes the paper.

2 RELATED WORK

Evolution is critical for the lifecycle of most software-based systems. Classifying the changes involved facilitates activities such as change propagation and change impact analysis. In model-driven software engineering (MDSE), several studies exist addressing evolution of models, meta-models, and DSLs. The co-evolution of related artifacts is also studied (Muctadir et al., 2023; Zhang and Strüber, 2024), but is not our focus here.

A catalog of change operators for meta-models and their impacts on instance models was presented by (Herrmannsdoerfer et al., 2011). This catalog was further improved by (Mengerink et al., 2016) claiming its incompleteness. Although these studies are conceptually similar to our work and focus on the structured description of potential changes, practically they are different. We concentrate on concrete syntax (i.e., grammar definitions), whereas the aforementioned studies focus on abstract syntax in the form of meta-models.

Various evolution scenarios of modeling languages and related artifacts were discussed by (Meyers and Vangheluwe, 2011). They also proposed a framework and algorithm for co-evolving these artifacts with the languages.

A systematic mapping study by (Thanhofer-Pilisch et al., 2017) focused on DSL evolution identifying 16 papers related to the evolution of DSL grammars. Most of them focus on developing and maintaining DSLs with a specific LWB. Through this study, we identified two papers (Tratt, 2008; Aschauer et al., 2010) relevant in the context of our work. (Aschauer et al., 2010) reported on their experience of an industry-academia collaboration, covering technical, domain-specific, and interpersonal complexities of industrial DSL evolution. (Tratt, 2008) presented a case-study showing DSL evolution along with changing requirements.

Taxonomising or classifying various change types is also well-researched for GPLs. These studies typically focus on the evolution of code written with GPLs as their syntax rarely changes. For example, (Sun et al., 2010) present a change impact analysis technique based on their proposed taxonomy classifying changes to object oriented programs (i.e., Java). We identified similar taxonomies in broader contexts. For example, (Lehnert et al., 2012) proposed a taxonomy of atomic changes for software systems. They also demonstrate how composite change types can be represented as a set of atomic ones.

This literature review shows that the complexities of artifact evolution are well-known in the software engineering research community. Researchers have classified these change possibilities into taxonomies, many of which focus on GPLs. In the MDSE domain, most classifications target meta-model changes and our search for taxonomies of DSL changes yielded limited results (Tratt, 2008; Aschauer et al., 2010).

3 TAXONOMY OF GRAMMAR CHANGES

In this section we first discuss the method we followed for developing the taxonomy and later present the taxonomy itself.

3.1 Taxomony Development

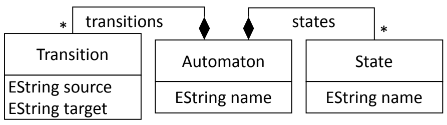
Our method for taxonomy development is influenced by the approach presented by (Nickerson et al., 2013) and broadly falls into the *intuitive* category of taxonomy development methods mentioned in that work. We created an initial taxonomy based on our understanding and experience with various textual LWBs. This initial version went through several iterations until the ending conditions were satisfied. The aim of each iteration was to identify overlapping, overlooked, or potentially unnecessary concepts, and address them by merging or removing existing concepts and adding new ones. For this purpose, we used the Xtext-based SysML-V2 grammar definition. It is open-source and its relatively large grammar definition files with a long change history make it suitable for our purpose. The ending criterion was that no modifications were made in the preceding iteration, essentially indicating that the taxonomy could describe all changes within the SysML-V2 repository.

3.2 Taxonomy of Grammar Change Operators

This section presents our taxonomy for grammar change operations. Table 1 highlights all the change operators, including an example for the state before and after the application of the respective change operator. Furthermore, it shows the impact on M2—the meta-model level—and on M1—the model level—and shows possible solutions. Using the example of Figure 1 we show the applicability of our taxonomy

```
1 grammar org.example.Automaton with common.Terminals   Xtext
2 generate domainmodel "http://www.example.org/Domainmodel"
3
4 Automaton: 'automaton' name=ID
5            '{'states=State* transitions=Transition* '}';
6 State:      'state' name=ID
7 Transition: 'transition' source=ID '->' target=ID
```

(a) A Xtext grammar defining the syntax of the automaton language.



(b) Meta-model produced for the abstract syntax of the grammar.

```
1 automaton PingPong {
2   state Ping state Pong
3   Ping -> Pong
4   Pong -> Ping
5 }
```

(c) A ping pong model conform to automaton grammar.

Figure 1: Running example of an automaton language.

with an automaton language specification in Xtext where Figure 1a shows the textual grammar, Figure 1b shows the meta-model induced by the grammar's abstract syntax, and Figure 1c shows a model conforming to the grammar. Xtext serves as a language workbench for developing textual DSLs (Erdweg et al., 2013). Developers can define the syntax of their DSLs within Xtext using a grammar definition language that employs Extended Backus-Naur Form expressions. Once a DSL is defined in Xtext, Xtext generates a full infrastructure that includes a parser, linker, type checker, compiler, and editor. Xtext grammars define a set of production rules with a left hand side providing the name of the nonterminal that the production is defining, and a right-hand side defining the sequence of nonterminal references and terminal symbols (cf. l. 4-7 in Figure 1a). Terminal symbols are indicated by surrounding quotation marks and nonterminal references are defined by a name followed by an equation sign and the name of the nonterminal in the grammar. Furthermore, referenced nonterminals can have a cardinality, e.g., the kleene star defining a zero to many cardinality (cf. l. 5).

Keyword. Since keywords do not become part of the meta-model, they only have impact on M1. We identify three different change operators: 1. Renaming the keyword impacts level M1 as all models containing the old keyword are no longer parseable. Instead the new keyword must be used. 2. When deleting the keyword, the model on M1 also becomes unparseable unless the former keyword is deleted. 3. Adding a keyword also impacts the M1 level in the way that the keyword must be used in the model. Concerning our example from Figure 1 we could change the keyword `"automaton"` (cf. Figure 1a l. 4) into simply `"aut"` which would impact the concrete syntax in the model Figure 1c requiring it to use the new keyword in order to conform to the grammar again.

Sequence of Nonterminals and Terminals. For a sequence of nonterminals and terminals, we could change its order. Since the order is not reflected in the meta-model it has no impact on M2. However, it has impact on the concrete syntax of models where the old order is not parseable anymore, and must be adjusted to the new order. Considering our example in Figure 1 we could change the order of `"state"` and `name` (cf. Figure 1a l. 6), meaning that both defined states in the model must be denoted as `Ping state` and `Pong state` (cf. Figure 1c l. 2).

Cardinality of Nonterminals. Change operators affecting the cardinality of nonterminals affect both M2 and M1. We distinguish between (1) changing the multiplicity from * (zero to many), to + (one to many), which has no effect on other concepts in M2, but on M1, since there has to be at least one instance of the nonterminal in the model, (2) deleting multiplicity impacting both M2 and M1, where on M2 the association between the metaclasses in the meta-model loses its multiplicity and M1 must have only one occurrence of the nonterminal, and (3) adding multiplicity which essentially has the reverse impact of deleting. In our example we could change the mulitplicity of `states` to + which would not affect the multiplicity relation from class `Automaton` to `Transition` (cf. Figure 1b), but would require at least one transition on M1 in the automaton model (cf. Figure 1c).

Nonterminal Definition. Change operators to the nonterminal definition can either (1) rename an existing definition, (2) delete an existing definition, or (3) add a new definition to the grammar. In the first case, this affects the reference to the nonterminal in the grammar and in most cases would yield an error when parsing the grammar since there is no nonterminal definition with the old name. Otherwise, this class would be disconnected from the class that referenced it before on M2. For instance, in Figure 1a renaming the nonterminal `Transition` (cf. l. 7) into `Trans` would make the reference in l. 5 invalid. The second

Table 1: The taxonomy for grammar change operators. Nonterminal (NT), Terminal (T), XT = supported by Xtext, MC = supported by MontiCore, NL = supported by Neverlang.

Grammar Concept	Operation	Example (before)	Example (after)	Impact on M2	Solution on M2	Impact on M1	Solution on M1	XT	MC	NL
Keyword	Rename	A = ”o” B	A = ”n” B	None	N/A	Old keyword not parseable	Change the keyword	✓	✓	✓
	Delete	A = ”o” B	A = B	None	N/A	Old keyword not parseable	Delete keyword	✓	✓	✓
	Add	A = B	A = ”n” B	None	N/A	New keyword missing	Add keyword	✓	✓	✓
Sequence of NT/T	Change order	A = B C	A = C B	None	N/A	Old order not parseable	Correct order	✓	✓	✓
Cardinality of NT	Change	A = B*	A = B+	None	N/A	Models without occurence of NT are not parseable	Add at least one occurence to model	✓	✓	✓
	Delete	A = B*	A = B	Association loses multiplicity	N/A	Models with multiple NT occurences unparseable	Use NT exactly once	✓	✓	✓
	Add	A = B	A = B+	Association gets muliplicity	N/A	None	N/A	✓	✓	✓
NT def. / production rule	Rename	A = B, B = ”b”	A = B, B’ = ”b”	References to class derived from renamed NT broken	Rename references to class derived from renamed NT	None	N/A	✓	✓	✓
	Delete	A = B, B = ”b”	A = B	References to class derived from deleted NT broken	Remove references to class derived from removed NT	Parts of model derived from deleted NT unparseable	Remove the unavailable parts	✓	✓	✓
	Add	A = B, B = ”b”	A = B, B = ”b”, C = ”c”	Disconnected class derived from added NT	Reference NT	None	N/A	✓	✓	✓
NT reference	Rename	A = b:B	A = b1:B	Association to NT is renamed	N/A	None	N/A	✓	✓	✓
	Delete	A = b:B c:C	A = c:C	Association to NT is not available	N/A	Right hand side (RHS) of NT unavailable	Remove the unavailable parts	✓	✓	✓
	Add	A = b:B	A = b:B c:C	Association to NT is added	N/A	Modified RHS makes legacy models incompatible	Add new syntax constructs	✓	✓	✓
Import	Rename	import G1	import G2	Import unresolvable, imported NTs no longer available	Revert change or redefine formerly imported NTs	None	N/A	✓	✓	(✓)
	Delete	import G1		Import unresolvable, imported NTs no longer available	Revert change or redefine formerly imported NTs	None	N/A	✓	✓	(✓)
	Add		import G1	Imported NTs are available for reuse	N/A	None	N/A	✓	✓	(✓)

case, where we delete an existing nonterminal definition, would lead to the same error as in the first case where all former references can no longer be resolved. When a new nonterminal is added, this nonterminal is expected to be referenced by another nonterminal. Otherwise, it is disconnected from the other classes in the derived meta-model, and therefore is not usable in the model. In our example, a new nonterminal `Guard` could enable modelers to define guards for transitions. However, if not referenced, e.g., by the production `Transition` the class would be added to the meta-model without any reference to the other classes, and hence would not be available in M1, too. For this last change operator, most language workbenches throw a warning.

Nonterminal Reference. When referencing nonterminal definitions, we can (1) rename these references, (2) delete them, or (3) add new ones. When we rename a nonterminal reference, the association in the meta-model is renamed. For instance, in Figure 1a consider renaming the nonterminal reference `states` (l. 5) to `statesList`. Then the association in

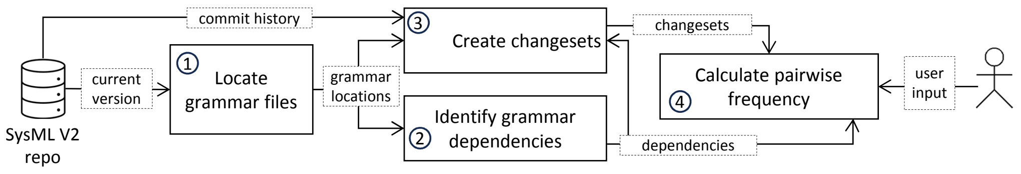


Figure 2: The process followed for the CIA. The circled numbers indicate the order of the corresponding steps.

Figure 1b between `Automaton` and `State` would be named accordingly. Deleting a reference in the grammar has the effect that the corresponding association is deleted in the meta-model, too, and also that the right-hand side of the referenced nonterminal is not available at M1 and, therefore, legacy models need to be adjusted to be parseable again. Adding a new nonterminal reference adds a new association to M2. Furthermore, the right-hand side of the newly referenced nonterminal needs to be instantiated in the right place in the models at M1.

Grammar Import. Importing grammars impacts M2 and enables the reuse of their nonterminals, facilitating reuse and modular language development. (1) Renaming the import, i.e., replacing the import by another one, entails that the formerly imported nonterminals are no longer available for use in the importing grammar. This can only be solved by importing another grammar with the same definitions or by redefinition of the formerly imported nonterminals. (2) Deleting an import entails the same impacts as changing it. (3) When adding a new import, the imported nonterminals become available for reference or in the importing grammar. For instance, in our automaton language (cf. Figure 1) we could import a grammar defining boolean expressions that we could leverage for guards on our transitions.

4 CASE STUDY

The frequent evolution of DSLs and co-evolution of related artifacts is a well-known research topic (Durisic et al., 2014; Hebig et al., 2017). The identification of impacted artifacts is a prerequisite for performing any manual or (semi-)automated co-evolution. CIA is defined as the identification of the consequences of proposed software changes (Bohner, 1996). In this case-study we analyze the impacts of changing DSL grammar definition on other related grammars and nonterminals. For this purpose, we choose to perform historical CIA (Li et al., 2013) on the Xtext-based grammar definitions for the SysML-V2. The goal of this case-study is to demonstrate an highly relevant example usage of the taxonomy in a real-world context. Here we use various Xtext specific terms, explained in (Efftinge and Spoenemann, 2024), omitting explanations for brevity. Additionally, we also explore the applicability of the taxonomy to two other LWBs.

4.1 Historical CIA on SysML-V2 Repository

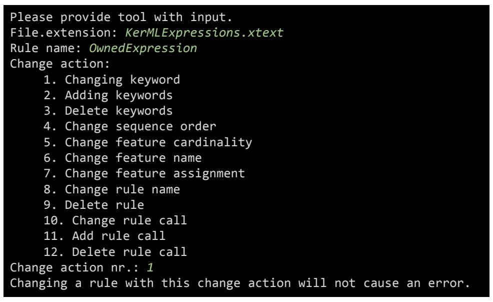
In our historical CIA implementation, we analyze the commit history of the SysML-V2 repository to determine the frequency of grammar files being modified concurrently. Using this information, we inform the user about the possible impact of changes made to a nonterminal of a certain grammar definition. Figure 2 depicts the steps we follow to perform the CIA. First, we locate all grammar definition files within the SysML-V2 repository. Afterwards, we look for two specific types of dependencies among these files: (1) A grammar definition exports an ecore meta-model that is imported by another grammar; (2) A grammar definition imports another. In the next step, we create changesets for all commits in the master branch of the repository. Such a set contains references to the changed files, the corresponding commit hash, and details of the changes, including the names of the rules that had undergone changes according to our taxonomy (cf. Section 3.2). The modifications made to the `import` statement are indicated using the term `$Grammar_Mixin$` in the change details. If related grammar files are changed in the same commit, their changesets are merged.

```
{
  "File": "KerMLExpressions.xtext",
  "Hash": "9b1216c0ec6a3ec1f07fe7092ea194239020a8d7",
  "Changeset": [ "BodyParameter” ],
  "External_File": "SysML.xtext",
  "External_Changeset": [ "Identification",
      "ConnectorEnd", "InterfaceEnd" ]
}
```

Figure 3: A changeset created from SysML-V2 repository.

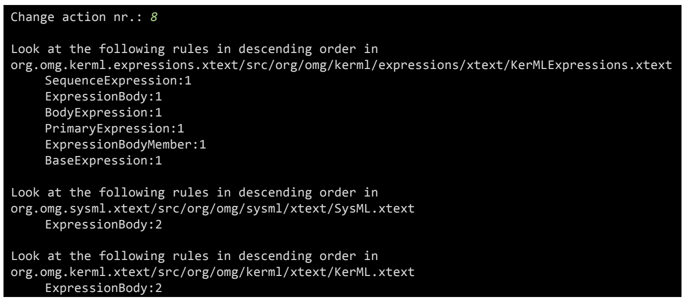
Figure 3 shows an example of such a changeset in JSON format where two dependent grammars `KerMLExpressions.xtext` and `SysML.xtext` were changed. This changeset also contains the names of the modified rules and specifies that the former grammar imports the latter.

These changesets are used to calculate the impacts of changing an existing grammar rule or nonterminal. To achieve this, the user needs to specify the name of



```
Please provide tool with input.
File.extension: KerMLExpressions.xtext
Rule name: OwnedExpression
Change action:
     1. Changing keyword
     2. Adding keywords
     3. Delete keywords
     4. Change sequence order
     5. Change feature cardinality
     6. Change feature name
     7. Change feature assignment
     8. Change rule name
     9. Delete rule
     10. Change rule call
     11. Add rule call
     12. Delete rule call
Change action nr.: 1
Changing a rule with this change action will not cause an error.
```

(a)



```
Change action nr.: 8

Look at the following rules in descending order in
org.omg.kerml.expressions.xtext/src/org/omg/kerml/expressions/xtext/KerMLExpressions.xtext
     SequenceExpression:1
     ExpressionBody:1
     BodyExpression:1
     PrimaryExpression:1
     ExpressionBodyMember:1
     BaseExpression:1

Look at the following rules in descending order in
org.omg.sysml.xtext/src/org/omg/sysml/xtext/SysML.xtext
     ExpressionBody:2

Look at the following rules in descending order in
org.omg.kerml.xtext/src/org/omg/kerml/xtext/KerML.xtext
     ExpressionBody:2
```

(b)

Figure 4: Console output from an example run of our historical CIA tool. Figure (a) shows that changing keyword at rule `OwnedExpression` in grammar `KerMLExpressions.xtext` will not cause any error. For the same grammar and rule, Figure (b) shows that changing the rule name can have internal and external consequences.

the Xtext grammar file and the rule name they want to modify. Using these inputs and the previously created changesets, we generate a list containing pairs of rule names along with the frequency of their concurrent modifications in the same commit. One of the elements of each pair of the list is the rule name specified by the user. This list, ordered decreasingly by frequency, provides insight into which other rules frequently undergo simultaneous modifications with the rule the user intends to change. The calculation of the pairwise frequency incorporates the aforementioned dependency information, ensuring that it covers grammar dependencies. However, this implementation heavily utilizes the commit history, making the method highly dependent on the availability and quality of these data.

An example console output from running our historical CIA tool is shown in Figure 4. During an execution, the user can provide the name of a grammar file, a rule name within it, and the type of change operator to apply to the rule. Figure 4a demonstrates an example run where the user wants to change a keyword of the rule `OwnedExpression` contained in grammar `KerMLExpressions.xtext`. Analysing historical data, the tool suggests that this is a safe change, as this has not historically affected any other artifacts. However, renaming the aforementioned rule is deemed unsafe due to various historical dependencies as shown in Figure 4b.

With this brief yet highly relevant case-study we demonstrate the applicability and usefulness of the taxonomy presented in Section 3, which plays a central role in the case-study. We use the taxonomy for creating the changesets, forming the basis for performing the historical analysis. Furthermore, the change action menu presented to the user while executing our CIA tool (cf. Figure 4a) is also based on our taxonomy.

4.2 Applicability to Other LWBs

MontiCore (Hölldobler et al., 2021). It is an LWB to define textual external DSLs with a grammar-based syntax definition. An example for a grammar specification is given in Figure 5.

```
1 grammar Automaton extends de.monticore.types.Types {          MCG
2   Automaton = "automaton" Name "{"
3               states:State* transitions:Transition* "}";
4   State = "state" Name;
5   Transition = source:Name "->" target:Name;
6 }
```

Figure 5: A MontiCore grammar (MCG) defining the syntax of an automaton language.

Concepts appearing are similar to those in Xtext. Via language inheritance, a grammar can extend existing grammars (cf. l. 1). The grammar then defines nonterminals that each have a left-hand and right-hand side. The left-hand side defines the name of the nonterminal, and the right-hand side defines a sequence of nonterminals and terminals (cf. ll. 2-3). Here keywords can be defined by quotation marks. Other nonterminals can be referenced, e.g., `State` (cf. l. 3, and l. 4). The * indicates that arbitrary many states and transitions can be defined. Since MontiCore includes at least all grammar concepts included in our taxonomy, we consider the latter to be applicable to MontiCore.

Neverlang (Vacchi and Cazzola, 2015). It is an LWB that enables modular development of textual external DSLs. Its syntax specification is also based on grammar rules that can be defined in so called modules. Figure 6 shows an example of a syntax for an automaton language similar to the one in Figure 5.

Similar to Xtext and MontiCore the syntax specification is rule based with the nonterminal name on the left-hand side and the sequence of nonterminals and terminals on the right-hand side. In contrast to Xtext and MontiCore, however, cardinalities can only be expressed by recursive rule statements (cf. ll. 5-7). Fur-

```
module Automaton {                                          NL
  reference syntax {
    Program    <-- "automaton"
                   Identifier "{" States Transitions "}";
    States  <-- State States;
    States  <-- State;
    State   <-- "state" Identifier;
    //…
  }
}
```

Figure 6: A Neverlang (NL) module defining the syntax of an automaton language.

thermore, modules in Neverlang do not import other grammars or modules themselves, but, instead, modules are composed in a higher level meta-language that defines a language out of a combination of different modules. Besides these different approaches, Neverlang supports all other grammar concepts that we identify in our taxonomy. As such Neverlang is compatible, too, and language engineers employing this LWB can benefit from our results.

5 CONCLUSION AND FUTURE WORK

This paper presents a taxonomy for grammar change operators and discusses their impact on the meta-model level M2 and modeling level M1. Furthermore, we provide solutions to the impacts that may produce conflicts on the respective levels. In our case study, we implemented a tool to demonstrate that our taxonomy can be leveraged to perform historical CIA and we argued why our taxonomy is applicable to grammar-based language workbenches beyond Xtext. In the future we plan to extend the taxonomy to recognize impacts of grammar changes on other language constituents, e.g., on well-formedness rules or code generators. Furthermore, we plan to extend it to a tool that can automatically derive dependency graphs from grammars that then can be leveraged for change propagation and can assist language engineers to maintain DSLs in an ever evolving system context.

ACKNOWLEDGEMENTS

This research was partially funded by NWO (the Dutch Research Council) under the NWO AES Perspectief program, project code P18-03 P3. The authors of the University of Stuttgart were supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) [grant number 441207927].

REFERENCES

Aschauer, T., Dauenhauer, G., and Pree, W. (2010). A modeling language's evolution driven by tight interaction between academia and industry. *Proceedings - International Conference on Software Engineering*, 2:49–58.

Bohner, S. A. (1996). Impact analysis in the software change process: a year 2000 perspective. In *1996 Proceedings of International Conference on Software Maintenance*, pages 42–51, USA. IEEE.

Butting, A., Pfeiffer, J., Rumpe, B., and Wortmann, A. (2020). A compositional framework for systematic modeling language reuse. In Syriani, E., Sahraoui, H. A., de Lara, J., and Abrahão, S., editors, *MoDELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020*, pages 35–46. ACM.

Durisic, D., Staron, M., Tichy, M., and Hansson, J. (2014). Evolution of Long-Term Industrial Meta-Models – An Automotive Case Study of AUTOSAR. In *EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 141–148.

Efftinge, S. and Spoenemann, M. (2024). Xtext - The Grammar Language — eclipse.dev. https://eclipse.dev/Xtext/documentation/index.html. [Accessed May 23, 2024].

Erdweg, S., van der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W. R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G. D. P., Molina, P. J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V. A., Visser, E., van der Vlist, K., Wachsmuth, G., and van der Woning, J. (2013). The state of the art in language workbenches - conclusions from the language workbench challenge. In Erwig, M., Paige, R. F., and Wyk, E. V., editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer.

Hebig, R., Khelladi, D. E., and Bendraou, R. (2017). Approaches to co-evolution of metamodels and models: A survey. *IEEE Transactions on Software Engineering*, 43:396–414.

Herrmannsdoerfer, M., Vermolen, S. D., and Wachsmuth, G. (2011). An extensive catalog of operators for the coupled evolution of metamodels and models. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6563 LNCS:163–182.

Hölldobler, K., Kautz, O., and Rumpe, B. (2021). *MontiCore Language Workbench and Library Handbook: Edition 2021*. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag.

Hölldobler, K., Michael, J., Ringert, J. O., Rumpe, B., and Wortmann, A. (2019). Innovations in model-based software and systems engineering. *Journal of Object Technology*, 18(1):1–60.

Hölldobler, K., Roth, A., Rumpe, B., and Wortmann, A. (2017). Advances in modeling language engineering. In Ouhammou, Y., Ivanovic, M., Abelló, A., and Bellatreche, L., editors, *Model and Data Engineering - 7th International Conference, MEDI 2017, Barcelona, Spain, October 4-6, 2017, Proceedings*, volume 10563 of *Lecture Notes in Computer Science*, pages 3–17. Springer.

Lehnert, S., ul-ann Farooq, Q., and Riebisch, M. (2012). A taxonomy of change types and its application in software evolution. In *2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems*, pages 98–107, Serbia. IEEE.

Li, B., Sun, X., Leung, H., and Zhang, S. (2013). A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23:613–646.

Mengerink, J., Serebrenik, A., Schiffelers, R., and van den Brand, M. (2016). A Complete Operator Library for DSL Evolution Specification. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 144–154, USA. IEEE.

Meyers, B. and Vangheluwe, H. (2011). A framework for evolution of modelling languages. *Science of Computer Programming*, 76:1223–1246.

Muctadir, H. M., König, L., Weber, T., Amrani, M., and Cleophas, L. (2023). Co-evolving meta-models and view types in view-based development. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 954–963, Sweden. IEEE.

Nickerson, R. C., Varshney, U., and Muntermann, J. (2013). A method for taxonomy development and its application in information systems. *European Journal of Information Systems*, 22:336–359.

Seidewitz, E., Miyashita, H., Wilson, M., de Koning, H. P., Ujhelyi, Z., Gomes, I., Schreiber, T., Bock, C., Grill, B., Zoltán, K., Marquez, S., Piers, W., Adavani, A., and Graf, A. (2023). SysML-v2-Pilot-Implementation. https://github.com/Systems-Modeling/SysML-v2-Pilot-Implementation. [Accessed October 2, 2024].

Sun, X., Li, B., Tao, C., Wen, W., and Zhang, S. (2010). Change impact analysis based on a taxonomy of change types. In *2010 IEEE 34th Annual Computer Software and Applications Conference*, pages 373–382, Korea (South). IEEE.

Thanhofer-Pilisch, J., Lang, A., Vierhauser, M., and Rabiser, R. (2017). A Systematic Mapping Study on DSL Evolution. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 149–156, Austria. IEEE.

Tratt, L. (2008). *Evolving a DSL Implementation*, pages 425–441. Springer Berlin Heidelberg, Berlin, Heidelberg.

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

Zhang, W. and Strüber, D. (2024). Tales from 1002 Repositories: Development and Evolution of Xtext-based DSLs on GitHub. In *SEAA'24: Euromicro Conference Series on Software Engineering and Advanced Applications*.