Int J Softw Tools Technol Transfer (2015) 17:601–626
DOI 10.1007/s10009-015-0387-9

SPLC 2013

Systematic synthesis of delta modeling languages

Arne Haber[1] **Katrin Hölldobler**[1] **Carsten Kolassa**[1] **Markus Look**[1]
Klaus Müller[1] **Bernhard Rumpe**[1] **Ina Schaefer**[2] **Christoph Schulze**[1]

Published online: 29 June 2015

Abstract Delta modeling is a modular, yet flexible approach to capture variability by explicitly representing differences between system variants or versions. The conceptual idea of delta modeling is language-independent. But, to apply delta modeling to a concrete language, either a generic transformation language has to be used or the corresponding delta language has to be manually developed for each considered base language. Generic languages and their tool support often lack readability and specific context condition checking, since they are unrelated to the base language. In this paper, we present a process that allows synthesizing a delta language from the grammar of a given base language. Our method relies on an automatically generated language extension that can be manually adapted to meet domain-specific needs. We illustrate our method using delta modeling on a textual variant of architecture diagrams. Furthermore, we evaluate our method using a comparative case study. This case study covers an architectural, a structural, and a behavioral language and compares the preexisting handwritten grammars to the generated grammars as well as the manually tailored grammars. This paper is an extension of Haber et al. (Proceedings of the 17th international software product line conference (SPLC'13), pp 22–31, 2013).

K. Hölldobler is supported by the DFG GK/1298 AlgoSyn.

✉ Markus Look
look@se-rwth.de
http://www.se-rwth.de/

Ina Schaefer
http://www.tu-bs.de/isf

[1] Software Engineering, RWTH Aachen University, Aachen, Germany

[2] Software Engineering and Automotive Informatics, TU Braunschweig, Brunswick, Germany

Keywords Delta modeling · Modeling · Language engineering · Domain specific languages · Generation · Software product line engineering

1 Introduction

Modeling is an important part of software development that allows focusing on essential system aspects in various development phases [12]. This holds for prescriptive modeling that aims at generating (parts of) software systems as well as descriptive modeling aiming at documentation or communication issues. Modern software systems are increasingly variable to adapt to varying user requirements or environment conditions. Software product line engineering [37] is a well-established approach for developing a set of systems with commonalities and variabilities. When a product line is constructed, all modeling techniques and languages used have to support the desired variability to allow a seamless integration into the development process.

There are three main ways to model variability within a software product line: annotative, compositional, and transformational variability modeling [21,57]. In this paper, we focus on delta modeling, a transformational variability modeling approach [20] which contains modular, yet flexible variability modeling concepts. In delta modeling, a set of diverse systems is represented by a designated core model and a set of deltas describing modifications to the core model. A particular product configuration is obtained by applying the changes specified in the deltas to the core model. A delta can add, remove, modify or replace elements of a model. Furthermore, other combined delta operations can be added to provide more convenient delta modeling possibilities for a developer, such as renaming or replacing elements.

[HHK+15] A. Haber, K. Hölldobler, C. Kolassa, M. Look, K. Müller, B. Rumpe, I. Schaefer, C. Schulze:
Systematic Synthesis of Delta Modeling Languages.
In: International Journal on Software Tools for Technology Transfer (STTT), 17/2015:601-626, Springer Berlin Heidelberg, October 2015.
www.se-rwth.de/publications

Delta modeling is a generic, language-independent concept. In order to use delta modeling for a particular modeling language, appropriate delta operations have to be defined for the considered language. Delta modeling has already been applied to the architecture description language MontiArc [22], to the programming language Java [43], to Class Diagrams [42], and to Simulink models [19]. Such delta languages can be used for individual activities as well as combined in any methodically useful way. Actually, delta modeling can be integrated in all stages of the software development process. In Fig. 1, we use the V-model as an exemplary software development process to demonstrate, where a delta language could be used. All stakeholders using a delta language, such as designers, developers, and testers, can benefit from the fact that the languages for describing variability within different artifacts in different phases of the development present themselves as homogeneous and closely related to the known languages as possible. Thus, by knowing a certain language, writing a delta for an artifact of the language becomes more intuitive. Additionally, by knowing one delta language, getting used to another one becomes also feasible due to its homogeneity.

The delta-oriented variability models in different development phases build on each other such that variability model structures in subsequent development phases can be derived from previous ones, as pointed out in [42].

However, to apply delta-modeling in a particular development phase, the used modeling or programming languages have to be extended by delta modeling concepts. The process of developing a delta language from a modeling or programming language is so far a manual process. Its result heavily depends on the knowledge and experience of the respective developer. This makes it hard to achieve homogeneity among different delta languages. However, for each modeling language that should be extended by a delta modeling language, very similar design steps have to be taken. Hence, the development of a delta language is very time consuming, since without a well-defined derivation method similar design decisions have to be made over and over again for every new delta language.

In order to alleviate this problem, in this article, we introduce an approach that allows us to systematically derive a delta language for any textual modeling (or programming) language. The main idea is to automatically generate an initial delta language based on a common delta language by applying a set of general derivation rules. The common delta language encapsulates the common concepts of delta modeling present in any delta language. The generated delta language can then be manually refined to meet domain-specific needs. Hence, the required effort for the manual design of a delta language is alleviated as only some adaptations have to be made. Furthermore, the generated and adapted delta languages are strongly related to each other as they have been derived by the same approach. This homogeneity of the delta languages leads to a simpler integration of different delta modeling languages in different development phases. In order to demonstrate the feasibility of our approach, we have implemented it using the language toolbench MontiCore [17,30–32]. This article is an extended version of [18] where we already presented the general concept to derive a delta modeling language from a given grammar. In this article, we enhance the presentation with additional examples and provide a evaluation of the presented approach by means of a comparative case study [41].

The paper is structured as follows: in Sect. 2, we illustrate the concept of delta modeling. Section 3 gives an overview of the language toolbench MontiCore. The method for synthesizing delta modeling languages is described in Sect. 4. In Sect. 5, we illustrate our method using a case example. Section 6 describes the implementation of our method. In Sect. 7, we evaluate our method using a comparative case study and discuss threats to validity in Sect. 8. Section 9 reviews related work and Sect. 10 concludes this paper.

2 Delta modeling on MontiArc

First, we illustrate the main concepts of delta modeling by the example of modeling variability in the MontiArc language [22].

MontiArc aims at modeling distributed interactive systems and allows to define components, their communication interfaces with incoming and outgoing ports, and their internal decomposition by instantiating and connecting further component definitions. It supports mechanisms like parametrization of components, generic port types, component extension by inheritance, invariant declarations, and comfort functions that should also be supported by a delta language.

MontiArc extends the base language Architectural Diagram (ArcD) that allows to define basic architectural model elements (components, ports, connections, see [22]). Afterwards, MontiArc adds simulation and behavior-specific elements like invariants, timed behavior, and comfort functionality to component models.

For the sake of simplicity only elements of the ArcD language are used in the following example. Nevertheless, all examples can be parsed by a MontiArc parser, too. In the following two different variants of a simplified car's interior light control system are used to demonstrate the basic idea.

The MontiArc component model in Listing 1 describes the internal structure of the basic variant. It consists of two incoming ports only: the light switch and the door state. The interior light shall be active if either the door is open or the light switch is turned on. The referenced component `a` of component type `Arbiter`, defined in another model and not discussed in more detail, is handling this decision.

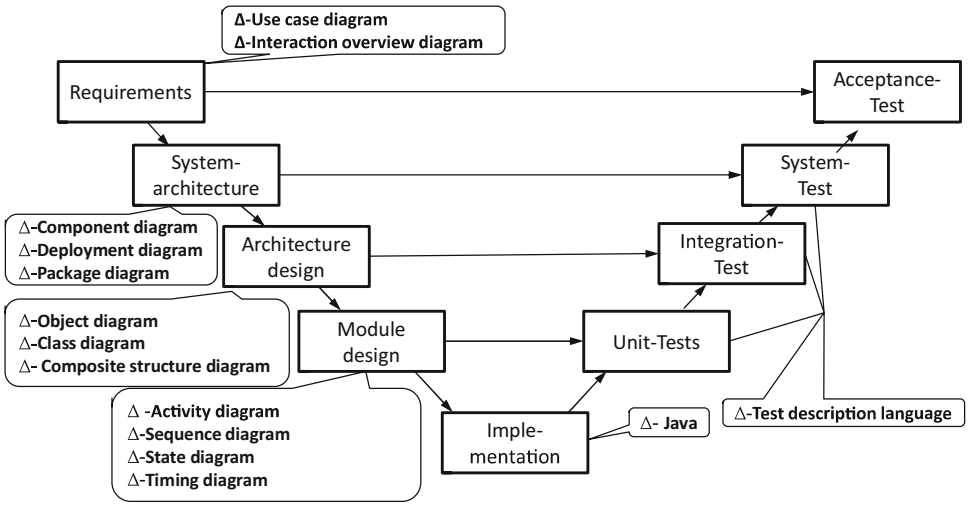

Fig. 1 Exemplary usage of different delta languages in the V-Model development process

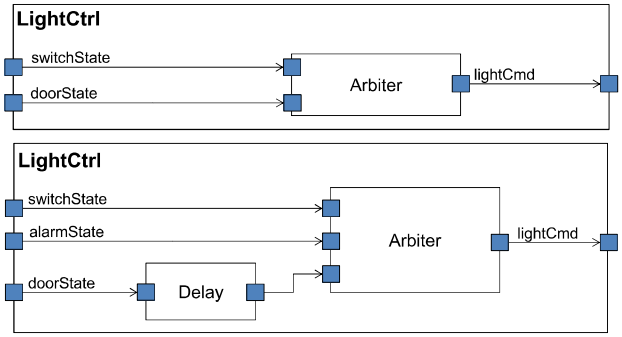

Fig. 2 Graphical representation of both interior light control system variants

```
1  component LightCtrl {
2    port
3      in Boolean switchState;
4      in Boolean doorState;
5
6    component Arbiter a;
7
8    connect switchState -> a.switchState;
9    connect doorState -> a.doorState;
10   connect a.lightCmd -> lightCmd;
11 }
```

Listing 1 Architectural diagram of the basic interior light control system variant

In a more advanced variant given in Listing 4 the interior light system additionally evaluates the alarm status of a corresponding alarm system. In case of an active alarm, the light shall always flash ignoring the switch and door state. In addition, the light shall stay active for a fixed delay after the door has been closed. This delay is performed by the referenced component `d` of type `Delay`. An equivalent graphical representation of both variants is depicted in Fig. 2.

In order to be able to represent variants of this component using deltas, we need a delta modeling language. The delta

```
1  delta DoorDelay {
2    modify component LightCtrl {
3      add component Delay d;
4
5      modify connection
6        [doorState -> a.doorState;] {
7          set target d.input;
8        }
9      connect d.output -> a.doorState;
10   }
11 }
```

Listing 2 Delta to add the door delay functionality

language should allow to add, remove or modify components, ports as well as connections and possibly other language concepts, if present. Listings 2 and 3 show instances of this delta language applied to derive the variant in Listing 4 by applying them to the component depicted in Listing 1. The first delta modifies the component `LightCtrl` (cf. Listing 2, l.2). It combines the following changes:

- it adds the component `d` of type `Delay` (cf. l.3), which realizes the required delay functionality.
- it modifies the `doorState` connection of the system to integrate `d`. This is accomplished by setting the target of the connection to the incoming port of `d` (cf. ll.5–8) and by connecting `d` to the `Arbiter` component `a` (cf. l.10).

The second delta modifies the component `LightCtrl` (cf. Listing 3, l.2) and the nested `Arbiter` component `a` (cf. l.5), which evaluates the actual light state based on given input. This delta combines several changes:

- an additional port is added to the `LightCtrl` component (cf. l.3) as well as to the `Arbiter` component (cf. l.6) to receive information about the current alarm state.

```
delta AlarmState {
  modify component LightCtrl {
    add port in Boolean alarmState;

    modify component a {
      add port in Boolean alarmState;

      add component Flash f;
      ...

      connect alarmState -> f.flashIn;
      ...
    }

    connect alarmState-> a.alarmState;
  }
}
```

Listing 3 Delta to add the alarm state evaluation functionality

```
component LightCtrl {
  port
    in Boolean switchState;
    in Boolean doorState;
    in Boolean alarmState;

  component Arbiter a;
  component Delay d;

  connect alarmState -> a.alarmState;
  connect switchState -> a.switchState;
  connect doorState -> d.input;
  connect d.output -> a.doorState;
  connect a.lightCmd -> lightCmd;
}
```

Listing 4 Product variant with alarm status evaluation and door delay functionality

- a connection between the newly introduced ports is established (cf. l.15).
- the internal structure of the `Arbiter` component is modified. A component realizing the flashing functionality is added (cf. l.8) and connected (cf. l.11). Additional modifications of the arbiter are left out for the sake of simplicity.

In our example we modify two components that are nested; hence, the modify statements are nested as well. We add a port to each of these two components; thus, we have two add statements one for each port. The order in which the two shown deltas are applied is not important, as both deltas do not depend on each other. Consequently, it is also possible to apply only one of the deltas.

One aspect, which has not been discussed so far, is the validation of delta models. As shown in Sect. 4.3, we use context conditions to check the correctness of our derived model and detect potential uncertainties via the context.

3 MontiCore language toolbench

The method of systematically synthesizing delta languages for textual modeling languages is based on the MontiCore language toolbench. We take languages defined as MontiCore grammars as input and produce delta languages that are also defined by a MontiCore grammar as result. In this section, we give a brief overview of MontiCore.

MontiCore supports the specification and generation of all relevant language processing artifacts for a specific textual language that is defined by a grammar similar to EBNF. Amongst other things, MontiCore generates the abstract and concrete syntax of a language, a lexer, a parser, and a set of runtime components, such as symbol tables and checkers for context conditions [30,56]. A MontiCore grammar is used to define the abstract as well as the concrete syntax of a language in a single artifact.

Listing 5 shows a simplified excerpt of the ArcD grammar defining the language that is used for modeling the architecture diagrams in Listings 1 and 4. A MontiCore grammar starts with the keyword `grammar` followed by the name of the grammar (cf. l.1) and contains a set of productions defining available language elements. Listing 5 shows four productions: `ArcElement` (cf. l.4), `ArcComponent` (cf. l.6), `ArcConnector` (cf. l.9), and `ArcInterface` (cf. l.13). `ArcElement` is an interface which is explained later in this section. `ArcComponent` defines the component itself, `ArcConnector` specifies connections between its ports and subcomponents, and `ArcInterface` specifies its ports.

A production consists of a nonterminal and a right-hand side (RHS) which specifies attributes and compositions within the abstract syntax tree (AST). As in EBNF, there might be terminals (surrounded by quotation marks (cf. l.7)) and nonterminals (cf. l.7) within the RHS. MontiCore allows to distinguish repeatedly used nonterminals by preceding the nonterminal with an identifier (cf. ll.10–11). For instance, in

```
grammar ArchitectureDiagram extends
    Types {

  interface ArcElement;

  ArcComponent implements ArcElement =
    "component" Name "{" ArcElement* "}";

  ArcConnector implements ArcElement =
    "connect" source:Name "->"
    targets:Name ("," targets:Name)* ";";

  ArcInterface implements ArcElement =
    "port" ...;

  ...
}
```

Listing 5 Simplified excerpt of the architectural diagram grammar

the production `ArcConnector` (cf. l.9), the nonterminal `Name` is preceded by the identifiers `source` (cf. l.10) and `targets` (cf. l.11). In this way, the source and the targets of the connection can be differentiated. We also have repetition (`A*` , `A+`), alternatives (`A | B`), and optionality (`A?`).

MontiCore also facilitates language reuse by supporting modularity concepts like, e.g., language inheritance and composition (not shown here) [34,44,56]. Language inheritance means that one or more existing grammars can be extended and refined by defining new grammar rules or redefining existing rules. This is denoted by the keyword `extends` followed by the names of the extended grammars (cf. ll.1–2). In this way, a language developer can focus on the differences between the existing languages and the new language. To ease the reusability and extensibility of languages, it is possible to define interface-nonterminals in MontiCore grammars. An interface-nonterminal can be used like any other nonterminal within the grammar (cf. l.7) and is introduced by the keyword `interface` (cf. l.4). This mechanism is an extended form of alternatives. Thus, the interface definition in l.4 can be interpreted as `ArcElement = ArcComponent | ArcConnector | ...`, where the RHS contains an alternative for every production that implements the interface. The language inheritance and interface concept in MontiCore is motivated by ob-ject-oriented inheritance and provides simple means to reuse and extend existing languages [29,34].

MontiCore also supports the definition and automatic checking of context conditions to verify that a model is well-formed. One simple context condition can, e.g., check whether the names of components within an architecture diagram are unique.

4 Derivation process

Based on MontiCore technology, we now introduce the method to derive a delta language for a given modeling language. The source language L needs to have the following properties:

- it needs to be textual
- it needs to be hierarchical.

There are three other properties that are not mandatory prerequisites but increase ease of use:

- the elements of L can be addressed via qualified names
- the qualified names of theses elements should not contain the character ".".

We use the "." character as a separator for qualified names. If qualified names can contain the "." character the user needs to configure a different character as the separator.

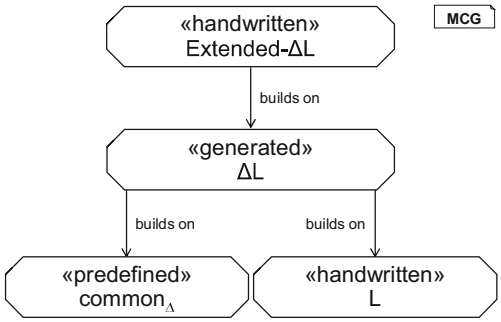

Fig. 3 Language hierarchy of concrete delta languages

The implementation of our approach relies on the language inheritance concepts of MontiCore. However, the method can be applied to every context-free grammar, since the used MontiCore concepts, such as grammar inheritance or interface productions, can easily be expressed likewise in EBNF. For instance, inheritance can be expressed by repeating the inherited productions and interface productions may be expressed by disjunctions.

Figure 3 shows the language hierarchy obtained when extending an existing source language L with delta modeling constructs. The basis is the abstract $common_\Delta$ language that predefines the overall structure of delta models. It additionally defines common delta operations and specifies how to identify elements in a model. The derived delta language ΔL extends this common language as well as the source language L.

This way all language elements of both languages are inherited and are available in the grammar of language ΔL. It is possible that the names of generated grammar rules collide with already existing names of grammar rules. This problem can be prevented by configuring a different prefix for the names of generated rules. The automatically derived language ΔL is already complete but can also be further refined manually to obtain a delta language $Extended\text{-}\Delta L$ that is tailored to domain specific needs.

4.1 Common delta constructs

The common structure for deltas is defined in the language $common_\Delta$ provided as a MontiCore grammar (Listing 6).

The syntactical structure of a delta is defined in ll.8–11. A delta has a unique name and consists of `Delta-Elements` (cf. l.11). This interface is implemented by productions that may be used directly within a delta. Each delta has an optional `ApplicationOrderConstraint` (*AOC*) (cf. l.10). An AOC is a logical expression over delta names, which restricts the execution order of deltas. It can be defined which deltas have to be applied before the current delta and which deltas must not be applied before. In the common grammar, `DeltaModify` (cf. ll.13–

```
1  // Elements that may be used directly
2  // within a delta model.
3  interface DeltaElement;
4
5  // Adds concrete syntax to modifies.
6  interface ScopeIdentifier;
7
8  Delta =
9    "delta" Name
10   ("after" ApplicationOrderConstraint)?
11   "{" elements:DeltaElement* "}";
12
13 DeltaModify implements
14     DeltaOperation, DeltaElement =
15   "modify" ScopeIdentifier
16   modelElement:ModelElementIdentifierPath
17   "{" DeltaOperation* "}";
18
19 // To identify model elements.
20 interface ModelElementIdentifier;
21
22 // Hierarchical path of MEIs.
23 ModelElementIdentifierPath =
24   parts:ModelElementIdentifier
25   ("." parts:ModelElementIdentifier)*;
26
27 // Default identifier: qualified name.
28 DefaultModelElementIdentifier implements
29   ModelElementIdentifier =
30   QualifiedModelElementName;
31
32 interface DeltaOperation;
33
34 // Operand of a delta operation.
35 interface DeltaOperand;
36
37 DeltaAdd implements DeltaOperand = "add";
38 DeltaSet implements DeltaOperand = "set";
39 DeltaRemove implements DeltaOperand
40     = "remove";
41
42 // Default remove operation.
43 DeltaRemoveOperation implements
44     DeltaOperation =
45   DeltaRemove
46   target:ModelElementIdentifierPath ";";
```

Listing 6 $common_{\Delta}$ MontiCore grammar

17) is the only production that implements the interface `DeltaElement`, therefore every `DeltaElement` is represented by a `DeltaModify`. This interface can later also be implemented in the *Extended*-ΔL-grammar to add further operations that may be used directly within a delta. The nonterminal `ScopeIdentifier` refers to an interface (cf. l.6) that is implemented by productions in the generated delta language and allows us to identify the model element type which is to be modified. The nonterminal named `modelElement` (cf. l.16) is used to define the context that is modified by the contained `DeltaOperations` (cf. l.17). Modify statements defined by the production `DeltaModify` may hierarchically contain further modify statements as this production implements the interface `DeltaOperation`.

A `ModelElementIdentifierPath` is needed to identify elements of the model. As depicted in Listing 6, it consists of dot-separated `ModelElementIdentifiers` named `parts` (cf. ll.23–25). Usually, models and their elements are hierarchically structured by a *contains* relation. Hence, the order of the parts has to reflect this hierarchical relation. Named model elements are typically identified by their name. Therefore, the default `ModelElementIdentifier` is a qualified name (cf. ll.28–30). Models also contain unnamed parts, e.g., connectors in MontiArc, respectively, ArcD. The `ModelElementIdentifier` interface is implemented in a concrete delta language for each unnamed model element that has to be identified within a delta.

The interface `DeltaOperation` shown in Listing 6 is implemented by delta operations that may be used within a modify statement (see l.17). Concrete operations must start with an operand `DeltaOperand` (cf. l.35) that defines the syntax of the concrete operation. Default operands are `add` for set-based elements of a model (cf. l.37), `set` for singular elements of a model (cf. l.38), and `remove` to delete elements from a set or to delete optional singular elements (cf. l.39). The default remove operation is given in ll. 43ff. The target of the operation is identified by a `ModelElementIdentifierPath` as explained above. Distinguishing between `DeltaOperation` and `DeltaOperand` allows us to generate a single production rule `DeltaOperation` for each nonterminal in the source language that represents all available modification operands at once.

4.2 Derivation rules

Based on the source language L and $common_{\Delta}$, we describe how to derive a delta language ΔL. For new nonterminals in ΔL, we use a composite name consisting of the name of the original nonterminal and the interface that is implemented, avoiding duplicate nonterminals. Within the following derivation rules, we use indices to represent this.

Addressing elements In the delta language, it should be possible to modify every model element given by the nonterminals of the concrete language. Thus, we need to provide an implementation of the interface `ModelElementIdentifier` for all nonterminals $N \in L$. With the following rules, we ensure that every nonterminal can be identified, either by the default production using a qualified name or the element itself. During the automatic generation of ΔL we consider an element as addressable if it contains a nonterminal `QualifiedName` with an identifier `name`.

1a *For every nonterminal N that can be identified by a qualified name, the default implementation of $common_\Delta$ is used to address the model element.*

1b *For every other nonterminal N, the concrete syntax of the corresponding model element enclosed in brackets is used for addressing it. Thus, for N, we introduce a new nonterminal ΔN_{MEI} and add a production of the form:*

$$\Delta N_{MEI}\ \texttt{implements}\ \texttt{ModelElementIdentifier} = \text{“[“}\ N\ \text{“]”}$$

Scope identifier The `ScopeIdentifier` interface of $common_\Delta$ is used to specify the element type that is addressed by the `ModelElementIdentifier`. With this, we are able to distinguish different model element types if they have the same `ModelElementIdentifier` but create different scopes for the modeled delta operations. At the same time, we are able to automatically create context conditions for checking matching identifiers and types. We reuse the nonterminal of L as concrete syntax of ΔL. With these kind of derivation rules, we are able to formulate modify statements that identify the model element and allow modifications within this scope.

2 *For every nonterminal $N \in L$, we introduce a new nonterminal ΔN_{SI} and generate a production of the form:*

$$\Delta N_{SI}\ \texttt{implements}\ \texttt{ScopeIdentifier} = \text{“}N\text{”}$$

Delta operation With this rule, we gain the ability to specify different delta operations inside a modify statement. The abstract grammar $common_\Delta$ defines the interfaces `DeltaOperation` and `DeltaOperand`. The implementation of these interfaces is needed for every nonterminal N since those are the elements that shall be modified inside a given scope.

3 *For every nonterminal $N \in L$, we introduce a new nonterminal ΔN_{DO} and generate an operation production of the form:*

$$\Delta N_{DO}\ \texttt{implements}\ \texttt{DeltaOperation} = \texttt{DeltaOperand}\ N$$

Multiple nonterminals This derivation rule is needed since we have to consider that nonterminals might be used more than once on the RHS of a production. In MontiCore, we distinguish those by identifiers preceding the nonterminals, as shown in Sect. 3. For ΔL we also need to distinguish these nonterminals because we would like to be able to modify them separately. We can reuse the identifiers and derive the productions for those operations. We add the nonterminal names as concrete syntax to the production to enable the distinction between the nonterminals inside the delta.

4 *For every nonterminal $N \in L$ used more than once on the RHS of a single production in L, we generate specific operation productions for each occurrence. For each identifier n_i of N, we introduce a new nonterminal Δn_{DO_i} and generate a production of the form:*

$$\Delta n_{DO_i}\ \texttt{implements}\ \texttt{DeltaOperation} = \texttt{DeltaOperand}\ \text{“}n_i\text{”}\ N$$

Delimiter addition Typically languages consist of block statements that hierarchically encapsulate other statements. Those block statements are delimited by an opening and a closing element. Inside block statements, there can be single statements that usually have a delimiter ending the statement. With a delta language, we can also modify parts of single line statements and not only complete single-line or block statements. Those parts usually have no delimiter. In this case, we add a delimiter to the corresponding delta production to achieve a uniform syntax of the delta language. For this reason, we analyze L and check if the nonterminal is either a block statement or a single line statement and has, therefore, a delimiter. Otherwise, we add a final delimiter to the nonterminals in ΔL.

5 *For every nonterminal $N \in L$ that is neither a block statement nor a single line statement with a line delimiter, we modify the operation production and append a delimiter.*

The derivation rules are sufficient to derive ΔL since they ensure that each nonterminal of L can be addressed to be modified, and additionally, every nonterminal can be used together with an operand inside a given scope. Thus, it is possible to modify every element by adding or removing other subelements. It should be noted that the rules use concepts provided by MontiCore but are not limited to them, since these concepts can be rewritten as other productions not using MontiCore concepts anymore. We show the application of these rules to the ArcD Language in the derivation process example in Sect. 5.

4.3 Context conditions

In addition to the derivation rules to create the delta language ΔL, we generate context conditions that provide some semantic checks for the delta language. The following enumeration provides context conditions that are automatically generated:

1. *A `ModelElementIdentifier` must reference an existing element.*

This can be done via resolving its qualified name and checking if there is an element with this name or via checking the complete concrete syntax of the element if used as an identifier for unnamed elements. This element may either exist directly in the core model or might already be added by a previous delta application.

2. *A* `ModelElementIdentifier` *must reference a model element that corresponds to its type given by the* `ScopeIdentifier`.
   This is not ensured by the concrete syntax of the language since most elements are addressed via qualified names which do not provide information about the type of the element.
3. *A* `ModelElementIdentifierPath` *must be valid in terms of its single concatenated elements.*
   While the previous context conditions focus on single elements, this context condition checks the path within the hierarchy of model elements.
4. *A* `DeltaOperation` *must be applicable within the scope of its surrounding modify statement.*
   This context condition can be inferred from the RHS of the production contained in L. Within the scope of the nonterminal on the LHS only operations affecting nonterminals from the RHS are allowed.
5. *A* `DeltaOperand` *must be applicable for its element.*
   Since we use the interface `DeltaOperand` to encapsulate the available delta operations, it is possible to use either the `add` operand or the `set` operand for a model element. While `add` should be used if there may be multiple possible elements, `set` should be used in case there is only one possible element. By using this an optional element is set whereas an existing element is changed. To ensure that `add` can only be used to add a new element to a collection and `set` can only be used for a single element, we use this context condition. It is derivable from the productions contained in L. Within the scope of the nonterminal used on the LHS, we can distinguish if a nonterminal on the RHS has a multiple cardinality and needs `add` as an operand, or if it has a single cardinality and thus needs `set` as an operand. This holds for multiple nonterminals that are distinguished by given variable names.
6. *An element that is mandatory for a certain model element must not be removed.*
   Typically there exist remove operations for every element within a language. But in case the production requires this element as a single mandatory element it is not allowed to specify a remove operation. If the element has a multiple cardinality or is optional this has to be possible nevertheless.
7. *An element that should be added must not exist.*
   This checking for avoidance of duplicates can be either done via the qualified name or via equality on the attribute level of the element.
8. *An element that should be removed must exist.*
   This check for existence of an element can be done in a similar way as the previous context condition.

4.4 Discussion

When automatically synthesizing a delta language ΔL from an existing source language L, we reuse concepts given in the language L. This also leads to the use of concepts of the abstract syntax of L which are typically hidden from the modeler who only knows the concrete syntax. But for specifying a modify statement, the delta modeler has to know the nonterminals of L as they become part of the concrete syntax of ΔL. For instance, we derive the keyword that identifies the scope of a modify operation in ΔL from the name of a terminal in L.

While the abstract and concrete syntax should typically be separated, we would like to present a method that automatically derives such a language ΔL. Hence, the above mentioned effect cannot be avoided completely.

In order to reduce this effect, it is possible to create a handwritten language $Extended\text{-}\Delta L$ that refines ΔL and overrides the productions defining these parts of the concrete syntax. To this end, we encapsulate the concrete syntax in own productions that can be overridden. Also those productions that contain, e.g., keywords that are not suitable and should be changed can be overridden.

Reusing parts of the abstract syntax of L and automatically synthesizing ΔL puts some requirements to the structure of L. For the automatic derivation of a delta language, the design of the abstract syntax is pretty important. The abstract syntax might contain folded or expanded productions that do not affect the concrete syntax of L. Especially the definition of productions and the use of nonterminals is important for deriving ΔL for the identification of model elements, since it affects the identification of model elements, the nesting of modify statements, and the feasible delta operations within a modify block. The possible path of navigation is given by the structure of the abstract syntax and might change if the abstract syntax changes. Therefore, it might be useful to restructure the grammar of L, e.g. by folding or unfolding nonterminals.

We avoid nondeterminism since we use a dynamic lookahead for parsing models of the context-free grammar. In addition, the resulting grammar ΔL is always non-left-recursive by construction since the derivation rules always introduce new unique nonterminals that are only used on the LHS of the productions and never on the RHS. The new nonterminals are based on the names of the original nonterminals to prevent name clashes.

MontiCore supports language reuse by grammar extension. Thus, the source language L might also be an extension of a parent language PL. For the derivation of the delta

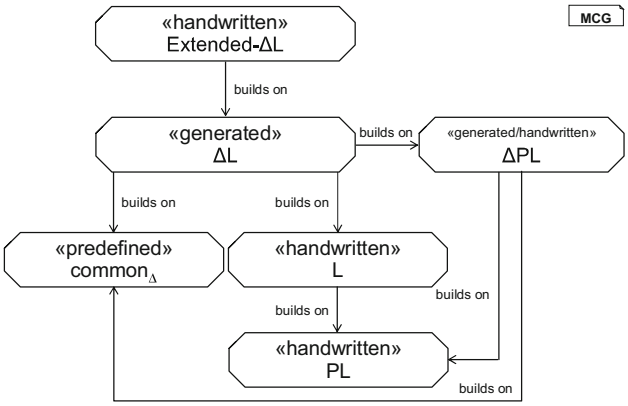

Fig. 4 Language hierarchy of concrete delta languages extended by PL and ΔPL

language from L, we only consider nonterminals defined directly in the language L and do not consider inherited nonterminals from parent languages yet. To handle language inheritance, we assume that for every parent language PL of L there also exists a delta language ΔPL, according to the language hierarchy shown in Fig. 4. The language ΔL builds upon ΔPL and can, therefore, also handle nonterminals defined in the super language.

5 Detailed example of the derivation process

In order to demonstrate our method, we use it to derive a delta language for the ArcD language introduced in Sect. 2. Afterwards, this language can be used to describe the deltas shown in Listings 2 and 3. In the following, we go through the derivation rules step by step, as described in Sect. 4.2, to show the contribution of each rule to the grammar of the delta language. The following steps are depicted in Fig. 5. Please note that a simplified grammar for ArcD is used to demonstrate the derivation method.

Our new delta language builds on the ArcD language. In MontiCore, this is done by extending the language (cf. Fig. 5, part A).

For the *first derivation rule*, we need to implement the `ModelElementIdentifier` interface for every nonterminal $N \in L$. In case the nonterminal N has a name we can use the default implementation in $common_{\Delta}$. An example for such a case is `ArcComponent`, because components have names in the ArcD language. The default implementation of `ModelElementIdentifier` is shown in Listing 6. If the nonterminal N has no name, which is the case for connections, we need to introduce a new nonterminal which we call `ArcConnectorIdentifier` that encloses the concrete syntax of an `ArcConnector` with square brackets. The corresponding part of the grammar is shown in part 1.

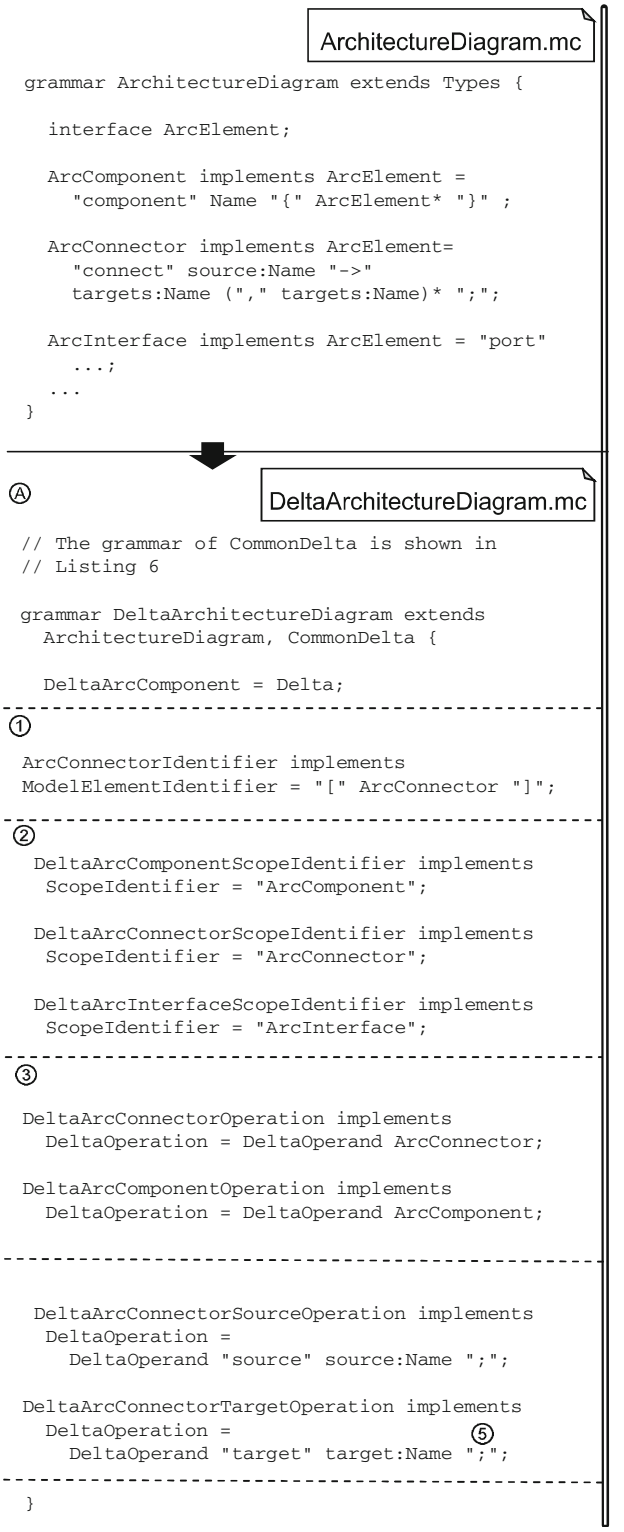

Fig. 5 Example for the application of the derivation rules

Using the *second derivation rule*, we derive the `ScopeIdentifiers` used in the modify statements, which are necessary to denote which architectural diagram grammar construct we want to modify. Part 2 shows the `Scope-`

```
1 grammar ExtendedDeltaArchitectureDiagram
2   extends DeltaArchitectureDiagram {
3     DeltaArcConnectorOperation implements
4       DeltaOperation = ArcConnector;
5 }
```

Listing 7 *Extended*-ΔL for the generated delta architectural diagram language L

`Identifiers` for `ArcComponents`, `ArcInterfaces`, and `ArcConnectors`.

With the *third derivation rule*, we specify the available operations for adding or removing connectors or components, see part 3. The delta operations are already defined in $common_{\Delta}$ (see Listing 6).

The original connector nonterminal consists of two name elements specifying source and target of a connection. As we would not be able to distinguish both we need to apply the *fourth derivation rule* after the third one and add the keywords ”target” and ”source” to the productions (see part 4).

The *fifth and last derivation rule* adds a delimiter to every statement that is neither a block statement nor a single line statement. In our case, this delimiter is a semicolon added by the fifth rule (cf. part 5).

However, the resulting concrete syntax of our generated delta architectural diagram language (Delta-ArcD$_{gen}$) does not conform to the expected syntax used in Sect. 2 yet. For instance, adding a new connection shall be indicated by the keyword `connect` instead of `add connect` (cf. Listing 2, l.9). According to the workflow presented in Sect. 6, we perform step (*A7*) and tailor the concrete syntax of DeltaArcD$_{gen}$ by creating an extended delta language *Extended*-DeltaArcD$_{gen}$. This is demonstrated in Listing 7. The shown grammar builds on the generated language `DeltaArchitectureDiagram` and redefines the production `DeltaArcConnectorOperation`. This way, the expected syntax for the connection modification is achieved.

The grammar of DeltaArcD$_{gen}$, which is partly shown in Fig. 5, in combination with its extension partly shown in Listing 7 can be used to generate a parser for our example in Listing 2.

The first and the last line are parsed using the `Delta` production (cf. Listing 6, l.8). Within the scope of such a `Delta`, only `DeltaElements` are allowed. A modify statement is a `DeltaElement`, i.e. the statement `modify component LightCtrl {...}` can be parsed using the production `DeltaModify` (cf. Listing 6, ll.13–17) which implements `DeltaOperation` and `DeltaElement`. Additionally, `DeltaModify` requires a `ScopeIdentifier` that is in our case the `DeltaArcComponentScopeIdentifier` from the ΔL grammar (cf. Fig. 5, part 2). Within the `DeltaModify` production multiple `DeltaOperations` are allowed (cf. Listing 6, l.17), in our examples these are the statements between ll.3 and 9 (Listing 2) and between ll.3 and 15 (Listing 3). They are either parsed using the `DeltaModify` production (cf. Listing 6, l.15), if they are modify statements, or by the newly generated productions `DeltaArcConnectorOperation` or `DeltaArcComponentOperation`, if they are `add` statements.

As the different `add` statements are pretty similar, we just describe how one of them is parsed in detail as the other ones are parsed similarly. We use the statement from Listing 2, l.3.

The whole statement is parsed using the `DeltaArcComponentOperation` production (cf. Fig. 5, part 3) which in turn uses the `ArcComponent` production from the architecture diagram grammar and the `DeltaOperand` nonterminal which is implemented by the `DeltaAdd` nonterminal.

The statement in Listing 3, l.8 is parsed exactly the same way, while the statement in l.3 is parsed similarly but using a `DeltaArcInterfaceOperation` production. The statements that start in ll.2 (Listing 2), 11 and 15 (Listing 3) are modify statements. They are parsed using the `DeltaModify` production (cf. Listing 6, ll.13–17).

The statement that starts in Listing 2, l.5 is special because it addresses a nonterminal that does not have a name. Therefore, we cannot use the default implementation of the `ModelElementIdentifier` interface but need to generate the `ArcConnectorIdentifier` production (cf. Fig. 5, part 1) which allows us to address connections using its complete syntax. As these statements are also parsed using the `DeltaModify` productions they can also include multiple `DeltaOperations`. In this case the `DeltaOperation` is a `DeltaArcConnectorTargetOperation`(cf. Fig. 5, parts 4 and 5). Apart from these two differences, it is parsed similarly to the `DeltaModify` already presented. This example shows that we can parse the DeltaArcD$_{gen}$ language using our newly generated grammar.

6 Implementation

In order to understand the delta generator one needs to understand the principles of the MontiCore language toolbench.

MontiCore is able to generate parsers from grammars which are used to create ASTs. These ASTs are then the basis for so called workflows, which are algorithms that work on these ASTs [17].

The `DeltaLSynthesisTool` contains MontiCore workflows such as `DeltaGeneratorToolWorkflow` and `DeltaCoCoToolWorkflow`. The relationship between the components of the delta generator is shown in Fig. 6 while their interaction is shown in Fig. 7 [in the following we reference the comments in both figures as *C* (*C*omponent dia-

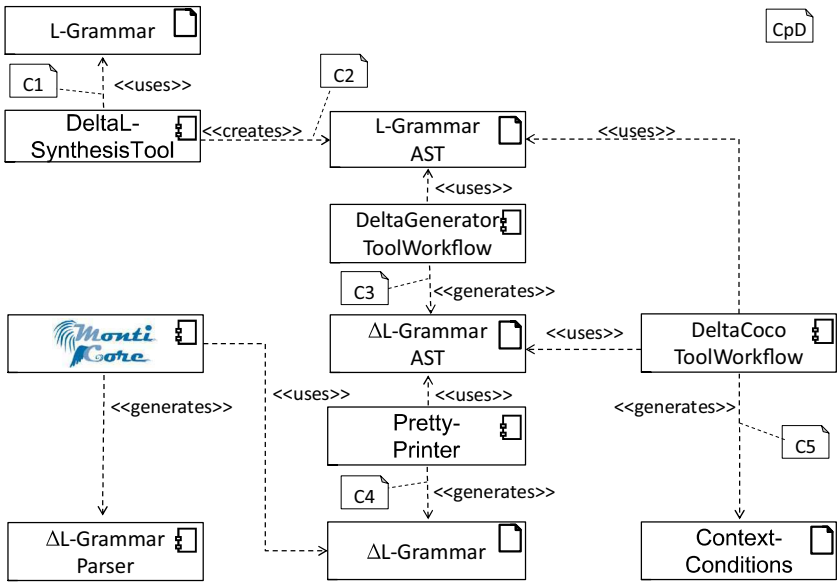

Fig. 6 Components of the delta generator

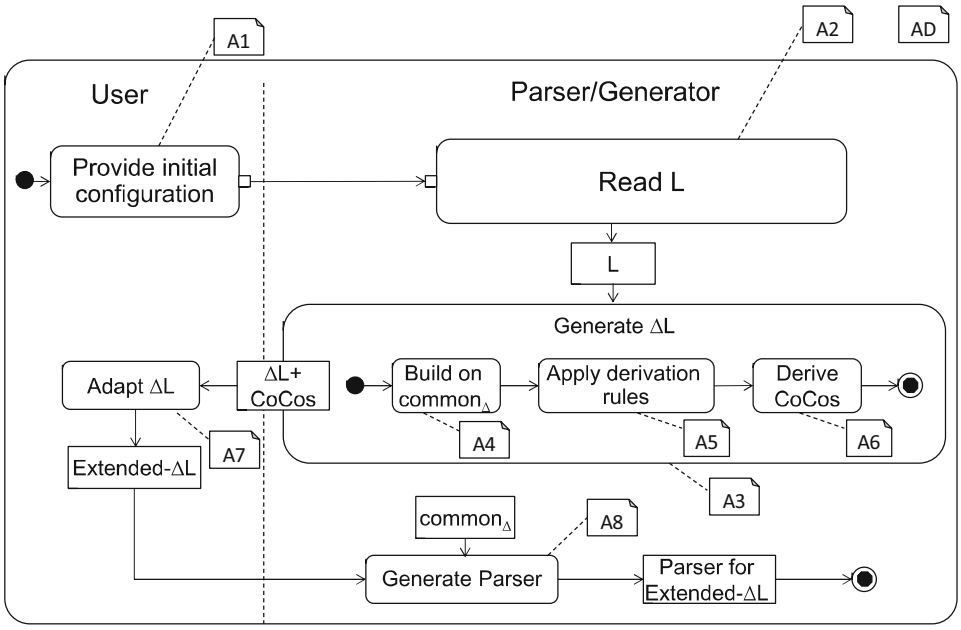

Fig. 7 Workflow of the automatic delta language derivation

gram) and *A* (*A*ctivity diagram), respectively]. Figure 6 shows which steps are performed by the user and which steps are performed by the generator/parser and, therefore, automatically. The delta generator is run via the MontiCore maven plugin which allows the execution of MontiCore tools within a maven managed build. This allows us to integrate the delta generator as part of a multistage setup.

The generation of a parser for ΔL from L is a two-stage process. In the first stage the user provides an initial configuration (*A1*) via the Project Object Model (POM-File) of the project. This configuration for example specifies which grammar L should be transformed to its delta counterpart. The delta generator parses (*A2*, *C1*) the grammar of L using MontiCore which creates the corresponding AST. The `DeltaGeneratorToolWorkflow` adds $common_\Delta$ as super grammar (*A4*, *C3*) to the AST of ΔL. Then a visitor is used to traverse the AST of L. The visitor also has a direct reference to the AST of the ΔL-grammar and generates new nodes for each node in the AST of L following the rules from Sect. 4 (*A5*, *C3*). If it visits a production it analyzes the production and creates new productions for nonterminals on the LHS according to rule 1–3 and for nonterminals on the RHS according to rule 4 of Sect. 4, while adhering to rule 5.

When the visitor finishes traversing the AST of L, the AST of ΔL is complete and written into a file using a PrettyPrinter

(*End of A3*, *C4*). The context conditions are generated by the `DeltaCoCoToolWorkflow` which uses the AST of L and ΔL as inputs (*A6*, *C5*). In the second stage MontiCore uses this ΔL-grammar to generate a parser for ΔL.

The user can now manually adapt ΔL by creating a subgrammar $Extended\text{-}\Delta L$ (*A7*). The manual adaption of a derived delta grammar is optional. It is useful, e.g., to tailor the syntax of the delta language. For example, ΔL may contain unneeded modify statements for language elements that should not be modifiable, or the syntax of an unnamed `ModelElementIdentifier` should be adjusted by introducing a new keyword. It is also possible to add more refined delta operations such as a replace operation. The names of nonterminals of the source language can be part of the derived delta language. This is, for example, the case when these nonterminals need to be addressed. Sometimes the names of these nonterminals are unintuitive and also need to be adapted while creating $Extended\text{-}\Delta L$.

7 Case study

In order to evaluate our method of generating ΔL_{gen} and manually extending it to $Extended\text{-}\Delta L_{gen}$, we designed a comparative case study.

An automatic approach has two potential advantages over the manual method:

- reduced effort
- human errors are eliminated.

These advantages only justify automation, if the resulting product is at least as expressive as the manual one and has no disadvantages that outweigh potential benefits. The different delta languages that have been created over time adapt the concept of delta languages to their specific needs. It is not possible to just dismiss these adaptions. We, therefore, need to compare languages created by human language designers to languages that are derived using our approach. We do this to show that our proposed manual tailoring method is able to create a language that reproduces the adaptions a human designer might apply to the concept of delta languages while still yielding a benefit over manually creating ΔL.

This leads to our main research question:

> *Main research question (MRQ)* Does our approach of generating ΔL_{gen} and manually extending it to $Extended\text{-}\Delta L_{gen}$ yield a benefit over manually creating ΔL_{hw}?

This benefit can be examined on two levels:

- the benefits of the method itself
- the benefits for the resulting language or at least the absence of disadvantages for the resulting language.

First we only look at the automatically derived language ΔL_{gen} without manual tailoring/extension. Our generated language directly contains all basic operations, namely `add`, `set`, `remove`, and `modify` (cf. Listing 6), that are also part of the handwritten languages in our case study. The handwritten languages contain additional operations beyond these basic operations.

We want to be able to express the same matters, in a semantic way, as the preexisting handwritten languages. We, therefore, need to check that these additional operations only add syntactic expressiveness and not semantic expressiveness like a new class of operations that our approach does not support. This leads to our first subresearch question:

> *Subresearch question 1 (SRQ1)* Has ΔL_{gen} at least the same semantic expressiveness as ΔL_{hw}?

In order to answer this research question we check whether the advanced operations of the handwritten languages can be represented as a sequence of the aforementioned basic operations and, therefore, do not add semantic expressiveness. The research question is deliberately constrained to ΔL_{gen}, since SRQ1 considers semantic expressiveness while the manual extension adds syntactic features.

Of course, syntactical equal expressiveness, enabled by the manual extension of the generated language, is also important. However, it is always possible to create an $Extended\text{-}\Delta L_{gen}$ that is identical to ΔL_{hw} since the sublanguage is able to overwrite and to introduce new productions. Thus, establishing the possibility to create the same syntactical expressiveness is inherently given. Nevertheless, such an extension is needed and important to create a customized and tailored language for a specific purpose with syntactic sugar or advanced syntactic operations.

To benefit from an automated approach, the effort of creating the manual extension should be less than directly creating the complete delta language by hand.

We use the complexity of the language as a proxy metric for the effort it would take to create it. The complexity of the language in our case is the number of productions and complexity of these productions.

If the extension of the language is less complex than the manually created one, a reduced effort in language design is to be expected. We again consider this as an indication of a benefit of the automated approach, leading to our second subresearch question:

> *Subresearch question 2 (SRQ2)* Is the complexity of $Extended\text{-}\Delta L_{gen}$ and, therefore, the effort to create it

smaller than the effort to create ΔL_{hw}, while being at least as syntactically expressive?

An implicit benefit of an automated approach is that it prevents human errors. We want to show that this implicit benefit is an actual benefit and that human errors actually exist in solely manually derived delta languages, even if they are already mature projects with quality assurance. The kind of human errors we will look at in the course of this case study are inconsistencies of the languages introduced by the human designer. We want to answer the following question:

Subresearch question 3 (SRQ3) Does $Extended\text{-}\Delta L_{gen}$ contain fewer inconsistencies than ΔL_{hw}?

Consistency in our case comprises two aspects. First, for each basic operation a reverse operation has to exist. This way for each model element kind that may be added to a model using an `add` operation, a `remove` operation has to exist. A `set` operation may be used to revert another `set` operation. Second, delta operations must not transform the model into an inconsistent state that cannot be repaired using further operations.

7.1 Case study design

We conduct a comparative case study [41] that compares a preexisting handwritten delta language ΔL_{hw} to a generated delta language ΔL_{gen} with a handwritten sublanguage $Extended\text{-}\Delta L_{gen}$ to answer the main research question. A broad application range is important for us we, therefore, chose an architectural, a structural, and a behavioral modeling language for the evaluation.

The languages have been derived using different methodologies and are based on different tools:

- Δ-MontiArc has not been created using our methodology but was developed using MontiCore.
- ΔCD_{hw} has been created using our methodology and is also using MontiCore.
- The ΔStatechart language is not MontiCore-based and was also not created using our methodology.

In the course of the case study we aim at answering our subresearch questions for each of the languages. We chose a comparative case study because each research question compares either ΔL_{gen} or $Extended\text{-}\Delta L_{gen}$ with ΔL_{hw}. The languages we chose differ from each other in various ways. We aim to explore if these differences have any impact on the answer of our research question.

To compare semantic expressiveness (SRQ1), we examine if a valid mapping for each advanced operation in ΔL_{hw} to a set of basic operations in ΔL_{gen} exists. By construction, all semantic basic operations of ΔL_{hw} are also available in ΔL_{gen}, because both delta languages correspond to the same source language. Thus, we explore the existence of a mapping from advanced to basic operations, which allows us to answer SRQ1. As the advanced operations are deviations of the standard construction of delta languages they are missing from ΔL_{gen} and we need to check whether they add semantic expressiveness as part of our case study.

In order to reason about the effort it takes to create $Extended\text{-}\Delta L_{gen}$ (SRQ2) in comparison to ΔL_{hw} (SRQ2), we count the number of productions of $Extended\text{-}\Delta L_{gen}$ and compare this with the number of productions of ΔL_{hw}. However, the number of productions is only representative if $Extended\text{-}\Delta L_{gen}$ accepts the same words as ΔL_{hw}. Thus, we additionally explore syntactical expressiveness and count the number of productions of $Extended\text{-}\Delta L_{gen}$ required to achieve this for answering SRQ2. Our comparison is only valid if the productions are of similar complexity; we discuss this in Sect. 8.1.

In order to be as syntactically expressive as ΔL_{hw}, $Extended\text{-}\Delta L_{gen}$ needs to be able to parse at least those words accepted by ΔL_{hw}. We obtain a test set of possible words, using a tool that is able to generate concrete words/models out of a given grammar. These models ensure that each derivation path within a grammar is used at least once. Internally the tool uses the Purdom-algorithm [38]. Thus, we can ensure that the generated models cover all paths of the resulting AST and provide a good coverage for testing expressiveness. The number of generated test words obviously corresponds to the size of the language in terms of the number of productions and the number of possible derivations. Apart from the generated words, we also reused existing test words for ΔL_{hw} that were available.

After generating the input words from ΔL_{hw}, we used the delta language generator to obtain ΔL_{gen}. Using the provided extension mechanism, we manually created $Extended\text{-}\Delta L_{gen}$. We iteratively adapted $Extended\text{-}\Delta L_{gen}$ until the generated parser was able to parse all words that were previously generated out of ΔL_{hw}. The generated parser of the handwritten language ΔL_{hw} is able to accept all generated and already existing words by construction. If $Extended\text{-}\Delta L_{gen}$ is adaptable to also be able to parse these words, we can conclude that $Extended\text{-}\Delta L_{gen}$ is at least as syntactically expressive as ΔL_{hw}.

While, in general, we can only state that $Extended\text{-}\Delta L_{gen}$ is at least as expressive as ΔL_{hw}, it is in some cases also possible to add additional test words that conform to ΔL_{gen} but not to ΔL_{hw}. In this case, $Extended\text{-}\Delta L_{gen}$ is syntactically more expressive.

For answering the second subresearch question, we count the necessary productions as a metric for complexity. We looked at the productions and operations required in the

extension and found out that they are of similar complexity as in ΔL_{hw} by manually comparing the productions. Thus, we use counting as a measure for the required effort to answer SRQ2.

For answering SRQ3, this test setup is also suited since we are able to closely examine in both grammars if the number of productions and operations deviate. This way, we are able to detect inconsistencies in terms of missing operations in ΔL_{hw}. Operations that may render a model to an inconsistent state are identified manually.

In the following subsections, we present our results and discuss them afterwards.

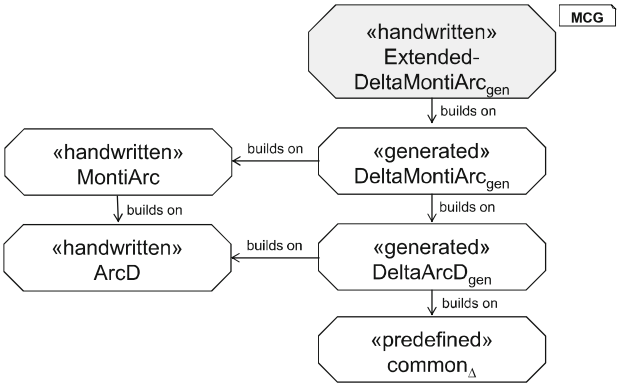

Fig. 8 Language hierarchy of the extended Delta-MontiArc grammar

7.1.1 MontiArc

In the first case we apply our method to MontiArc [22], which has been introduced in Sect. 2. For MontiArc, we use 67 manually written and 48 generated models to evaluate syntactical expressiveness of the involved languages.

Δ-MontiArc has not been designed according to the method described in Sect. 4 so that this language can be regarded as a highly customized delta language.

The following language productions of Δ-MontiArc add advanced delta operations:

1. *ModifyParamStatement* A special `modify` statement allows to modify parameter assignments of subcomponents.
2. *RemoveUnreachablesStatement* A `remove unreachables` operation allows to remove unconnected ports of components.
3. *ReplaceStatement* A `replace` statement allows to replace a subcomponent with a type-compatible subcomponent.
4. *RenameStatement* A set of `rename` operations to rename ports, subcomponents, parameters and invariants.
5. *AutoConStatement*
   - The operation `expand autoconnect` locally or globally removes the `autoconnect` statement and makes connectors explicit. This way they may be explicitly modified by further delta operations.
   - The reverse operation `introduce autoconnect` locally or globally removes explicit connections that may be also created by an autoconnect and later on introduces an autoconnect statement. This operation is realized together with the `expand autoconnect` operation in a single language production.

Additionally, some delta operations in Δ-MontiArc have a syntax that is not compliant to the syntax proposed in Sect. 4. These are realized in the following productions:

1. *AddArcElementStatement* This production provides the regular `add` operation for all architectural elements. It additionally provides an add operation for connectors with an adjusted syntax. Instead of `add connect` a simple `connect` is used to add a new connector. This corresponds to the original MontiArc syntax for creating a connection. As this production realizes both, default and adjusted syntax, it is not counted here.
2. *AddConfigStatement* Configuration elements from the MontiArc language are also directly added without the `add` operation prefix.
3. *DisconnectStatement* Instead of `remove connect` a more intuitive `disconnect` is used to remove a connector.
4. *AddParamStatement* Configuration parameters are added with `add param Type Name` instead of `add Type Name`.
5. *AddTypeParamStatement* Type parameters are added accordingly with `add typeparam`.

The language hierarchy of the generator based languages is depicted in Fig. 8, it reflects MontiArc's language hierarchy.

7.1.2 Class diagrams

In the following, we present how our method can be applied to a textual class diagram language. This language is part of the UML/P [39,40,44] language family.

ΔCD_{hw} was developed during a practical course with five undergraduate student participants. The method proposed in Sect. 4 was given to the students beforehand as a reference implementation manual. The task of the practical course was to develop DeltaCD based on the class diagram language, to directly adapt the concrete syntax of the language to a more intuitive one and to directly introduce new refactoring language features.

Based on the method, the students first developed a language containing the scope identifiers `CDAssociation`, `CDMethod`, `CDAttribute`, `CDConstructor`, and `CDEnumconstant` which are in fact the names of the nonterminals of the class diagram language as proposed in Sect. 4. Since these keywords are less intuitive we decided to change them to `association`, `method`, `attribute`, `constructor`, and `enumconstant`.

Furthermore, we added for some `add` and `remove` operations an additional keyword, such that `remove id` to remove an attribute with name `id` becomes `remove attribute id`.

In total we added four remove keywords, four add keywords, and two keywords for both adding and removing. Also a simplification was done, by adding a keyword that enables changing the direction of an association. Thus `setleftToRight−>` and `setrightToLeft<−` resulting from our method can be alternatively written as `set associationdirection −>` and `set associationdirection <−`, respectively.

Additionally, the functionality of the language was extended by new operands, based on the refactorings defined by Fowler et al. [11]:

1. *rename* Renames a given element to the new name. Possible elements are classes, interfaces, attributes, method parameters, enums, enum constants, associations, and methods.
2. *move* Moves a given attribute or method to another class.
3. *merge* Merges two given classes into a new class.
4. *extract* Extracts a given set of attributes or methods into a new class or a new interface. Specifying an extraction as a subclass or a superclass of the element the attributes or methods are taken from is also possible.
5. *pullup* Pulls a given attribute or method from multiple subclasses up into a common superclass.
6. *pushdown* In contrast to pullup, pushdown pushes an attribute or a method from a superclass to a subclass.

The applicability of these language features is checked via context conditions which are not part of this case study. These context conditions check if the pulled up attribute, the source classes, and the target class all exist and that all source classes have such an attribute with the same type.

Summing up, we made

- 15 syntax adaptions, such as changing and adding keywords,
- one simplification, i.e. aggregated keywords,
- six language feature additions, such as the new operand `move`.

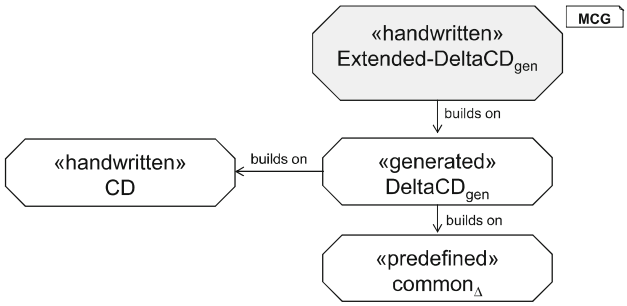

Fig. 9 Language hierarchy of the extended Delta-CD grammar

The concrete language hierarchy of DeltaCD$_{gen}$ is shown in Fig. 9. We take the existing class diagram grammar CD as an input to generate the delta language DeltaCD$_{gen}$ (ΔL_{gen}) for class diagrams. Following the proposed method DeltaCD$_{gen}$ builds on a common predefined grammar common$_{\Delta}$ which already defines the basic operation to add and remove elements of a language. To adjust the concrete syntax and to introduce new features we manually created an extended delta language $Extended$-DeltaCD$_{gen}$ that builds upon the generated. For the evaluation we follow the experimental setup. We use DeltaCD, created during the practical course, as a reference and create $Extended$-DeltaCD$_{gen}$ ($Extended$-ΔL_{gen}) such that all models conforming to DeltaCD can be accepted by $Extended$-DeltaCD$_{gen}$. As input models we used 156 manually written and 93 generated models.

7.1.3 Statecharts

In the last case of our case study, we evaluate how our method can be applied to a textual statechart language, called Statecharx. The handwritten language we use for comparison is neither created using our method nor using MontiCore. The language is defined by an XText [63] grammar and describes a subset of UML statecharts.

For the case study, we took the corresponding delta language, independently created in XText. We used no handwritten models for the statechart language and eight generated ones. This delta language is inherently different from our proposed method and uses some concepts that are not part of our language yet. At the moment, the language is able to define the modification of a statechart within a delta. Within this modify block it is possible to add and remove states, transitions, and their labels.

In order to evaluate our method with Statecharx, we manually rewrote the given XText grammar to a MontiCore grammar. This can be done straightforward since XText also uses a textual grammar based on EBNF. While most changes are syntactical, we adapted the referencing mechanism used in XText to simple names. This referencing is used to spec-

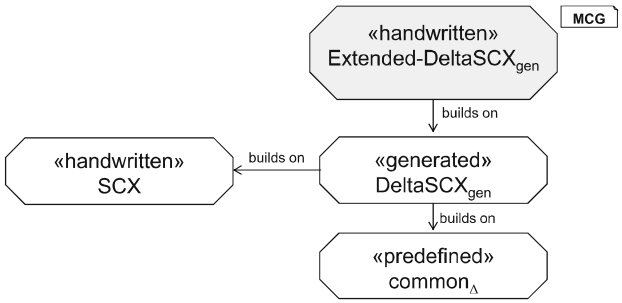

Fig. 10 Language hierarchy of the extended Delta-SCX grammar

ify within the grammar that a certain name references the name of another model element. In the case study, we simply use names but do not ensure anymore that the used names in a concrete model are real references since we want to syntactically evaluate the expressiveness. As a result, we got two MontiCore grammars, i.e. the language SCX based on the Statecharx language and the language DeltaSCX (ΔL_{hw}) based on the handwritten delta language in XText.

As depicted in Fig. 10, we applied our generator to SCX and obtained the generated delta language DeltaSCX$_{gen}$ (ΔL_{gen}) which is based on our method. We created the handwritten language extension *Extended*-DeltaSCX$_{gen}$ (*Extended*-ΔL_{gen}) of DeltaSCX$_{gen}$ to mirror the changes in the handwritten DeltaSCX. With these languages at hand we followed our evaluation method and measured the results in the same way as in the previous cases.

7.2 Results

The results of the presented case study motivated in Sect. 7.1 are summed up in Tables 1 and 2 as well as Figs. 11 and 12. Defining a mapping from advanced semantic operations to basic operations allows us to answer SRQ1, and comparing the number of productions and operations allows us to answer SRQ2 and SRQ3.

The left columns of Tables 1 and 2 list all advanced operations available in Δ-MontiArc and DeltaCD, respectively. Since DeltaSCX does not contain any advanced operations, it is not considered. The right column contains a sequence of basic operations that allow to express the same semantic operation. These basic operations are defined by DeltaMontiArc$_{gen}$ and DeltaCD$_{gen}$, respectively. The depicted syntax is partially simplified for reasons of clarity and, thus, may not directly correspond to the syntax of the generated delta languages.

Figure 11 compares the number and kind of productions in the original delta languages, the generated delta languages and the handwritten extensions that build upon the generated delta languages. It is distinguished between handwritten productions and generated, respectively, provided productions.

Table 1 Mapping from advanced Δ-MontiArc semantic operations to basic operations in DeltaMontiArc$_{gen}$

Advanced operation	Mapping to basic operations

Modify component c ($x = 5$, $y = 6$)	`modify component c {`
	`  remove arguments;`
	`  add argument x = 5;`
	`  add argument y = 6;`
	`}`

Remove unreachables	for all port p in unconnected {
	`  remove port p;`
	}

Replace component c1 with c2	`remove component c1;` `add component c2;`
Rename port p1 as p2	`remove port p1;` `add port p2`

Expand autoconnect	for all connectors c
	  in autoconnect {
	`  add connector c;`
	}
	`remove autoconnect;`

Introduce autoconnect	`add autoconnect;`
	for all connectors c
	  in autoconnect {
	`  remove connector c;`
	}

Handwritten productions are further distinguished in productions that add advanced delta operations (*Adv.*), productions that are created to customize the syntax of delta operations (*Syntax*), and other productions that either add operations being conform to the default syntax proposed in Sect. 4 or productions that structure the language (*Other*). The sum of these three handwritten production kinds is given in column $\sum_{hw}$. The number of generated and provided productions is summed up in column *Gen/Prov*. Provided productions are inherited from an existing super-grammar that has not been handwritten by the delta language designer (e.g. language common$_{\Delta}$). The total number of productions per language is given in column $\sum$. These results are also visualized in the stacked bar chart on the RHS of the figure. Please note that the contained sums $\sum_{hw}$ and $\sum$ are not part of the bars depicted.

It can be seen that the generated languages DeltaMontiArc$_{gen}$, DeltaCD$_{gen}$, and DeltaSCX$_{gen}$ do not contain any handwritten productions. Their total number of productions is determined by the number of generated productions only (73, 76 and 30, respectively).

Δ-MontiArc has the same number of productions that add advanced operations than *Extended*-DeltaMontiArc$_{gen}$ while the latter contains two more syntax-adjusting productions. Δ-MontiArc contains six other productions, *Extended*-DeltaMontiArc$_{gen}$ has none. Δ-MontiArc contains zero generated or provided productions, DeltaMon-

Table 2 Mapping from advanced DeltaCD semantic operations to basic operations in DeltaCD$_{gen}$

Advanced operation	Mapping to basic operations
Rename attribute a1 to a2	`remove attribute a1;` `add attribute a2;`

Move attribute a1 from class c1 to class c2	`modify class c1 {`
	`  remove attribute a1;`
	`}`
	`modify class c2 {`
	`  add attribute int a1;`
	`}`

Merge class c1, c2 to class c3	`add class c3 {`
	`  for all elements e in c1, c2 {`
	`  e;`
	`  }`
	`}`
	`remove class c1, c2;`

extract class c1 {
attribute a1;
method m1;
} as class c2;
	`modify class c1 {`
	`  remove attribute a1;`
	`  remove method m1;`
	`}`
	`add class c2 {`
	`  int a1; void m1();`
	`}`

Pullup attribute a1 from class c2, c3 to class c1	`modify class c2, c3 {`
	`  remove attribute a1;`
	`}`
	`modify class c1 {`
	`  add attribute int a1;`
	`}`

Pushdown attribute a1 from class c1 to class c2, c3	`modify class c1 {`
	`  remove attribute a1;`
	`}`
	`modify class c2, c3 {`
	`  add attribute int a1;`
	`}`

tiArc$_{gen}$ and *Extended*-DeltaMontiArc$_{gen}$ have the same number of generated and provided productions.

Extended-DeltaCD$_{gen}$ has fewer handwritten productions of all three kinds than DeltaCD. DeltaCD$_{gen}$ and *Extended*-DeltaCD$_{gen}$ have the same number of generated and provided productions. DeltaCD contains nine provided productions as it builds upon the (earlier version of) common$_{\Delta}$.

Extended-DeltaSCX$_{gen}$ contains seven handwritten productions and 30 generated while DeltaSCX contains only eight handwritten productions. As shown in Fig. 11, DeltaSCX has two syntactical productions and six other productions whereas all productions of *Extended*-DeltaSCX$_{gen}$ are syntactical.

Figure 12 contains the number of delta operations available in the given languages. These are modify, add, remove, set, and advanced operations. The results are visualized in a stacked bar chart on the RHS. Again the column $\sum$ is not part of the depicted bars. Please note that sometimes semantically equivalent operations are available with alternative syntax. These operations are counted only once.

It can be seen that Δ-MontiArc consists of three times less operations than DeltaMontiArc$_{gen}$. Especially the number of modify, remove, and set operations is considerably smaller. *Extended*-DeltaMontiArc$_{gen}$ only contains six operations more than DeltaMontiArc$_{gen}$. These are one modify operation and five advanced operations.

DeltaCD$_{gen}$ consists of about two times as many operations than DeltaCD. It offers significantly more remove operations, the number of other operations only slightly differs. *Extended*-DeltaCD$_{gen}$ contains seven more advanced operations than DeltaCD$_{gen}$.

DeltaSCX$_{gen}$ contains about four times as many operations as DeltaSCX. DeltaSCX does not contain any set operation and significantly less remove operations. It also contains slightly less add operations. *Extended*-DeltaSCX$_{gen}$ has only one additional modify operation compared to DeltaSCX$_{gen}$.

7.3 Discussion

Within the discussion, we answer the three research questions posed in Sect. 7. As explained previously, SRQ1 addresses the semantic expressiveness of the languages, SRQ2 addresses the complexity of the manual adaption compared to the manual created language, and SRQ3 addresses the consistency of the languages. In the following, we answer these questions for each evaluated language separately. Then, we sum up the results for all examined languages within a concluding discussion section to answer the main research question.

7.3.1 MontiArc

Within this subsection we will compare the handwritten delta language Δ-MontiArc to the generated delta language DeltaMontiArc$_{gen}$ and its handwritten extension *Extended*-DeltaMontiArc$_{gen}$. The results of this comparison are used to answer the subresearch questions.

SRQ1 Every advanced operation provided by Δ-MontiArc may be mapped to a set of basic operations that are semantically equivalent, as already shown in Table 1. The `modify` operation that allows to modify parameters of subcomponents is mapped to a modify block that removes the used arguments and adds the modified arguments again. Operation `remove unreachables` is mapped to a function that iterates over the set of unconnected ports and removes them accordingly. A `replace` operation is mapped to a removal of the replaced subcomponent and adding the substitute afterwards. Operation `rename` that is used to rename, e.g., ports, is handled the same way by removing the old port

Language	Adv.	Syn.	Oth.	$\sum_{hw}$	Gen/Prov	$\sum$
Δ-MontiArc	5	4	6	15	0	15
DeltaMontiArc$_{gen}$	n/a	n/a	n/a	n/a	73	73
Ext.-DeltaMontiArc$_{gen}$	5	6	0	11	73	84
DeltaCD	19	43	9	71	9	80
DeltaCD$_{gen}$	n/a	n/a	n/a	n/a	76	76
Ext.−DeltaCD$_{gen}$	19	17	0	36	76	112
DeltaSCX	0	2	6	8	0	8
DeltaSCX$_{gen}$	n/a	n/a	n/a	n/a	30	30
Ext.-DeltaSCX$_{gen}$	0	7	0	7	30	37

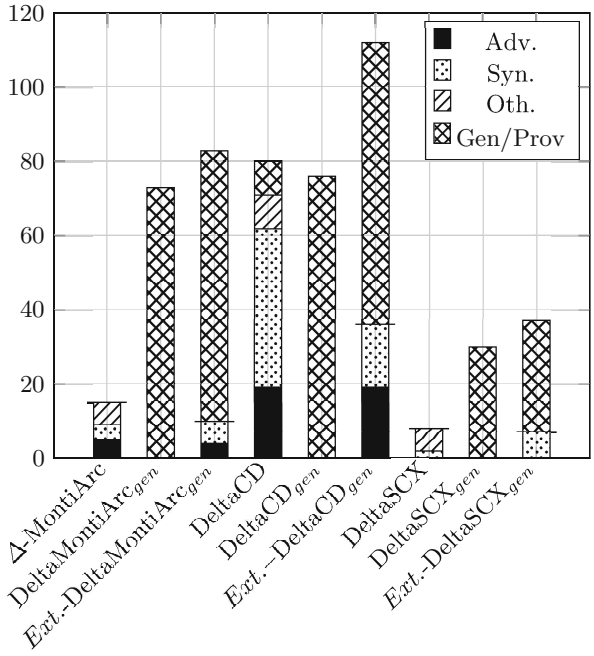

Fig. 11 Comparison of production statistics of original, generated, and extended delta languages

Language	mod.	add	set	rem.	adv.	$\sum$
Δ-MontiArc	1	10	0	6	6	23
DeltaMontiArc$_{gen}$	15	14	21	35	n/a	85
Ext.-DeltaMontiArc$_{gen}$	15	14	21	35	6	91
DeltaCD	9	13	15	13	8	58
DeltaCD$_{gen}$	12	24	29	53	n/a	118
Ext.−DeltaCD$_{gen}$	12	24	29	53	8	126
DeltaSCX	2	3	0	3	0	8
DeltaSCX$_{gen}$	5	4	9	13	n/a	31
Ext.-DeltaSCX$_{gen}$	6	4	9	13	0	32

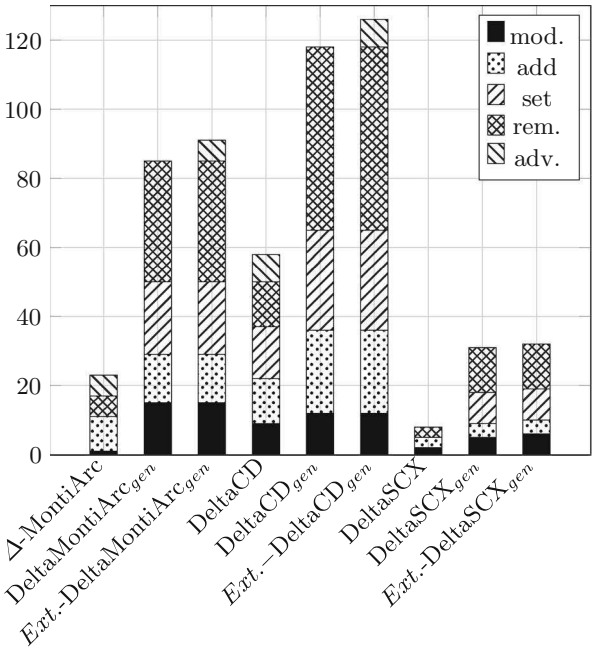

Fig. 12 Comparison of available delta operations in original, generated, and extended delta languages

and adding a new port with the target name and the same type and direction. `Expand autoconnect` is realized by iterating over all connectors that are created by the used autoconnect statement and explicitly adding them to the current scope. Afterwards, the original autoconnect statement is removed. Finally, operation `introduce autoconnect` is mapped vice versa by first adding an autoconnect statement and then removing all ports that are implicitly created by this statement.

DeltaMontiArc$_{gen}$ offers operations that are not available in Δ-MontiArc. All MontiArc model elements may be added and removed in a delta. Additionally, DeltaMontiArc$_{gen}$ allows fine-grained modifications of all model elements. As summed up in the Table depicted in Fig. 12, it offers about

3.7 times more semantic operations than Δ-MontiArc (85 vs. 23). These fine-grained operations are an additional feature that our approach generates while manually derived languages are often developed iteratively. Implementing the most important operations first makes it difficult to implement fine grained operations later.

All basic operations of Δ-MontiArc are per construction also available in DeltaMontiArc$_{gen}$ and all its advanced operations may be mapped to semantically equivalent operations in DeltaMontiArc$_{gen}$ as shown in Table 1. Thus, DeltaMontiArc$_{gen}$ has at least the same semantic expressiveness as Δ-MontiArc.

SRQ2 As a prerequisite of SRQ2, we first ensure syntactic expressiveness to be able to investigate the complexity of the languages.

Extended-DeltaMontiArc$_{gen}$ is able to accept all deltas that are accepted by Δ-MontiArc. These are the manually developed deltas as well as the delta models that have been artificially generated by MontiCore (see Sect. 7.1). Besides the syntax defined by Δ-MontiArc, *Extended*-DeltaMontiArc$_{gen}$ is also able to accept deltas given in the syntax defined by DeltaMontiArc$_{gen}$. These models may not be accepted by Δ-MontiArc. This way a MontiArc modeler who is aware of the method described in Sect. 4 and does not know the syntax of Δ-MontiArc may also use *Extended*-DeltaMontiArc$_{gen}$. Thus, *Extended*-DeltaMontiArc$_{gen}$ is syntactically more expressive than Δ-MontiArc.

Extended-DeltaMontiArc$_{gen}$ has four handwritten productions less than Δ-MontiArc. It contains productions to achieve the same syntax and the same advanced operations that are described in Sect. 7.1.1. The complexity of these productions does not differ, because they are identically constructed. Two additional simple syntax-adjusting productions are needed that are not part of Δ-MontiArc. Structural productions are completely inherited from the generated language DeltaMontiArc$_{gen}$ and the provided common$_{\Delta}$ language. Therefore, the grammar of *Extended*-DeltaMontiArc$_{gen}$ is less complex than the grammar of Δ-MontiArc, which also means that it is less effort to create *Extended*-DeltaMontiArc$_{gen}$ than creating DeltaMontiArc$_{hw}$.

SRQ3 Following the evaluation, Δ-MontiArc contains some inconsistencies in its design. It is possible to add some model elements, while there is no remove operation. As depicted in Fig. 12, Δ-MontiArc provides ten `add` operations, but only six `remove` operations. In addition, it is not possible to modify the parent type of a component in Δ-MontiArc. Since DeltaMontiArc$_{gen}$ is generated according to the method described in Sect. 4, all `add` or `set` operations have their corresponding `remove` operation. Additionally, it is possible to modify a component's parent type. Thus, Δ-MontiArc contains more inconsistencies than DeltaMontiArc$_{gen}$ and *Extended*-DeltaMontiArc$_{gen}$.

7.3.2 Class diagrams

Within this subsection we will compare the handwritten delta language DeltaCD to the generated delta language DeltaCD$_{gen}$ and its handwritten extension *Extended*-DeltaCD$_{gen}$. The results of this comparison are used to answer the subresearch questions, while the main research question will be answered in a concluding discussion.

SRQ1 As before, for answering SRQ1, we map the advanced operations of DeltaCD to basic operations in DeltaCD$_{gen}$. Table 2 shows that the six advanced operations can be successfully mapped to a sequence of basic operations. The `rename` operation can be mapped to removing the element and adding it again with a different name. This has to be done within the same modify scope. The `move` operation corresponds to removing an element in one modify scope and adding it in a different. Furthermore, the `merge` operation merges two elements into a new third one. Thus, it corresponds to adding the union of both elements to the newly added element. The last three operations `extract`, `pullup` and `pushdown` are quite similar. Extracting corresponds to adding certain elements to a completely independent new element and removing them from the old element, while pulling up and pushing down assume that the new element is located up, or down, respectively, the inheritance hierarchy.

Additionally, DeltaCD$_{gen}$ also contains more fine grained operations for adding and removing elements. By construction all basic operations of DeltaCD are contained in DeltaCD$_{gen}$, and we are able to map all contained advanced operations to basic ones. Thus, DeltaCD$_{gen}$ has at least the same semantic expressiveness as DeltaCD.

SRQ2 As a prerequisite of SRQ2, we again ensure syntactic expressiveness to be able to investigate the complexity of the language.

Extended-DeltaCD$_{gen}$ is able to accept all models that can be accepted by DeltaCD, thus, it is at least as syntactically expressive as the handwritten language. It is even more syntactically expressive since it also accepts models that can be accepted only by DeltaCD$_{gen}$, but not by DeltaCD.

DeltaCD contains 19 productions that are needed to add advanced functionality, as shown in Fig. 11. Of course, these productions are also required in the manual adaption. Apart from those, *Extended*-DeltaCD$_{gen}$ contains 17 productions that adjust the syntax of the language, as described in Sect. 7.1.2. Due to the modular design of DeltaCD$_{gen}$, these productions are easy to implement since they all have the same structure and simply exchange a single keyword.

The overall number of productions that have to be implemented manually is cut in half taking the advanced operations into account, while the number of productions even dropped to a quarter not considering the advanced operations. Thus, the manual adaption of DeltaCD$_{gen}$ to get *Extended*-DeltaCD$_{gen}$ is less effort than writing the grammar of DeltaCD$_{hw}$ in a solely manual approach.

SRQ3 Even though the creation of the handwritten grammar DeltaCD during the practical course also followed our method, there is a difference in terms of consistency between the handwritten and the generated delta language DeltaCD$_{gen}$. As shown in Fig. 12, DeltaCD$_{gen}$ contains considerably more remove operations than DeltaCD. The method to derive delta languages ensures a `remove` operation for each `add` or `set` operation, e.g. removing the name of a class. In DeltaCD, some remove operations were unintentionally left out. Additionally, DeltaCD does not contain `modify`, `add` or `set` operations for every nonterminal of the class diagram grammar. There are only three additional `modify`, but 11 more `add` and 14 more `set` operations contained in DeltaCD$_{gen}$ than in DeltaCD. This difference between the number of `modify` and `add`, `set` operations, respectively, is due to the multiple use of nonterminals on the RHS of a production which in turn leads to a single `modify` but multiple `add` or `remove` operations. Thus, regarding the third subresearch question DeltaCD$_{gen}$ and *Extended*-DeltaCD$_{gen}$ have fewer inconsistencies as DeltaCD$_{hw}$. Apart from being more consistent, the manual creation of the delta language is error prone and certain operations may be left out or accidentally forgotten while the automated generation guarantees that the language is complete (no possible delta operation is missing) and consistent.

7.3.3 Statecharts

As a representative of the behavioral languages we chose a statechart language as a base language and used the handwritten delta language DeltaSCX for comparison to the generated language DeltaSCX$_{gen}$ and its handwritten extension *Extended*-DeltaSCX$_{gen}$. We first answer the three posed subresearch questions and answer the main research question in a concluding discussion.

SRQ1 As shown before, we only map the advanced operations of ΔL_{hw} to the basic operations of ΔL_{gen}. Since the basic operations of ΔL_{hw} are implicitly contained in DeltaSCX$_{gen}$ by construction and DeltaSCX does not provide any advanced operations, SRQ1 can be answered positively.

SRQ2 As a prerequisite of SRQ2, we also ensure syntactic expressiveness to be able to investigate the complexity of the language.

Extended-Delta-SCX$_{gen}$ accepts all delta models that have been generated for DeltaSCX. It additionally accepts the delta models generated for DeltaSCX$_{gen}$ which are not accepted by DeltaSCX. Thus, it is syntactically more expressive than DeltaSCX.

Extended-DeltaSCX$_{gen}$ contains one handwritten production less than DeltaSCX. Because DeltaSCX has not been developed according to the method described in Sect. 4, *Extended*-DeltaSCX$_{gen}$ only contains simple syntax-adjusting productions. In contrast, DeltaSCX contains only two syntax-defining productions while containing six productions that define the structure of a delta. *Extended*-DeltaSCX$_{gen}$ does not need those, since they are already provided by DeltaSCX$_{gen}$. Thus, apart from having nearly the same number of productions, we consider it to be less effort to write the extension to create *Extended*-DeltaSCX$_{gen}$ than creating DeltaSCX$_{hw}$.

SRQ3 DeltaSCX allows to remove a state that is marked as initial, but is not able to adjust the according marker. Therefore, removing this state renders the statechart invalid. Further, DeltaSCX does not foresee nesting of modify operations and, thus, is only able to address the top-level elements of statecharts, such as labels, states, and transitions. Therefore, elements contained in either of these productions cannot be modified. These features are directly provided by DeltaSCX$_{gen}$ because the automated approach ensures completeness. Which in turn makes *Extended*-DeltaSCX$_{gen}$ consistent (e.g. every add operation has a corresponding remove operation) while DeltaSCX$_{hw}$ misses operations.

7.3.4 Summary

The three presented cases have shown that our method can be applied to behavioral, structural, as well as architectural languages. The statechart case has shown that it is also applicable to independently defined textual languages whose definition may be transformed into a MontiCore grammar.

For all three cases, all three subresearch questions are answered in favor of our approach. Hence, we are able to answer our main research question positively as well. Thus, we conclude that our method, in fact, provides a suitable starting point for engineering delta modeling languages. For the cases in our case study we have been able to derive languages with less effort that are more consistent while being syntactically and semantically at least as expressive as the manually developed delta languages. Consequently, using our method for all involved languages in a development process leads to homogeneous delta languages.

8 Threats to validity

The literature mentions four main threats to empirical research in software engineering [62]. These are threats to conclusion validity, internal validity, construction validity, and external validity.

8.1 Threats to conclusion validity

The *conclusion validity* concerns the question: "Does the treatment/change we introduced have a statistically significant effect on the outcome we measure?" [10]. Thus the conclusion validity concerns the degree to which the drawn conclusions are reasonable [52].

The sample size of our case study is too small to be able to draw conclusions that are statistically representative and go beyond the cases we showed in the study. The conclusions can, therefore, only be made for the cases we have studied. Two people checked the metrics we measured (e.g. number of grammar rules) independent from each other and came to the exact same numbers. The differences in those metrics have been big enough to be significant for the respective cases.

For answering SRQ1 we check whether the advanced operations of ΔL_{hw} can be mapped to basic operations of ΔL_{gen}.

In SRQ2, we check that the effort to create the extension $Extended\text{-}\Delta L_{gen}$ is smaller than the effort to create ΔL_{hw} in the cases we examine. For this purpose, we compare the number of productions that need to be written manually for $Extended\text{-}\Delta L_{gen}$ to the number of productions of ΔL_{hw}. We can show that the extension has less productions in all our cases.

In order to use the number of productions as a metric for complexity, the individual productions in ΔL_{gen} need to be less or equally complex than in ΔL_{hw}. The complexity of adding advanced delta operations to $Extended\text{-}\Delta L_{gen}$ is not significantly different from the complexity of writing them directly as part of ΔL_{hw}, while refining the syntax of an already existing operation is less complex since the keywords of the concrete syntax can be adapted easily in $Extended\text{-}\Delta L_{gen}$. Due to this fact, the number of productions is a conservative measure for effort as the number of productions in $Extended\text{-}\Delta L_{gen}$ is smaller than for ΔL_{hw}, while the productions are less complex.

In SRQ3, we evaluate whether ΔL_{hw} is inconsistent when compared with $Extended\text{-}\Delta L_{gen}$. The derivation process guarantees that for each add or set operation a corresponding remove operation will be provided. But we need to show that this is an actual benefit over handwritten languages. The handwritten languages in our evaluation have been developed with quality control measures. We can show that there are still inconsistencies compared to generated languages.

Apart from the inconsistencies in the base operations there can be inconsistencies that are introduced by advanced operations. It is still possible to introduce these inconsistencies into the handwritten extension of the generated grammar. This, however, is more unlikely, because this extension builds already upon a consistent language and it is less complex (SRQ2) than the completely handwritten grammar.

8.2 Threats to internal validity

Internal validity refers to the question "Did the treatment/change we introduced cause the effects on the outcome? Can other factors also have had an effect?" [10].

The metrics we used in the evaluation are the number of productions and a comparison of the semantics of the handwritten delta language ΔL_{hw}, the generated delta language ΔL_{gen}, and the handwritten sublanguage $Extended\text{-}\Delta L_{gen}$. Consequently, the metrics solely depend on ΔL_{hw}, ΔL_{gen}, and $Extended\text{-}\Delta L_{gen}$. As ΔL_{gen} was generated by our method, the only other factors that can influence the results are factors, which influence the quality of the grammars for ΔL_{hw}, the language we compare the generated language to. If the handwritten grammar ΔL_{hw} has many flaws, the generated grammar looks very good in comparison, while the results would be different with another handwritten grammar that is more complete and has less inconsistencies.

The quality of ΔL_{hw} depends on the experience the language developers had in the cases we used for comparison. The languages ΔL_{hw} and $Extended\text{-}\Delta L_{gen}$ have been developed by people with a comparably high level of experience in language development or have been supervised by experienced language developers to provide the required balance. Therefore, we think that the results of the comparison are valid.

In order to further mitigate this problem, we chose three handwritten languages as the cases for our comparative case study that already have been utilized in research or industrial projects. We, therefore, assume that these languages do not have a particularly bad quality.

Due to these arguments, we are convinced that our comparative case study has a high internal validity.

8.3 Threats to construction validity

Construction validity answers the question: "Does the treatment correspond to the actual cause we are interested in?" [10].

To answer SRQ1, we check whether we can express the advanced operations of ΔL_{hw} using base operations of ΔL_{gen}. The expressiveness of these semantic base operations and the semantic expressiveness of a language are directly related.

The results of these comparisons only depend on the case from the compare group, in our case language L and ΔL_{hw}, and the case that uses our method. The only human influence is the comparing and matching of the semantic operations of ΔL_{hw} and ΔL_{gen}, or in other words in comparing the compare case with the one that uses our method. It is unlikely that an error occurred here as errors would only occur in one direction and that is being not able to express an advanced operation using base operations although it is possible. To compare the effort of creating $Extended\text{-}\Delta L_{gen}$ (SRQ2), the number of its productions is counted. As already discussed in Sect. 7.1, the complexity of each single production can be neglected under the assumption that complex operations are realized similarly. Therefore, we reviewed each production individually to ensure that they are comparable in their complexity. Under this precondition, a direct relation between complexity and the number of productions can be established.

For SRQ3, a delta language is inconsistent, if it provides operations, which cannot be undone by other operations or which transform the model to an unrecoverable invalid state. In relation to the first case the number of add and remove operations are compared as for each add operation a corresponding counterpart has to be given to perform an undo. The second case is a more general indicator for an inconsistency, which cannot be measured easily. Therefore, each grammar has been analyzed manually, and all detected inconsistencies are discussed in Sect. 7.3 in detail to ensure a correct interpretation. Nevertheless, it cannot be ensured that all inconsistencies could be detected.

We provided several arguments that our treatment corresponds to the actual cause and, therefore, we believe our study has a high degree of construction validity.

We can show that even mature languages have inconsistencies that have been introduced through human error while generated ones do not show these inconsistencies. Manually extending a generated language can introduce new inconsistencies (human error) but this is less likely as the extension is less complex as shown in SRQ2.

8.4 Threats to external validity

The *external validity* concerns the following question: "Is the cause and effect relationship we have shown valid in other situations?" [10]. Therefore, external validity or transferability decides whether our results can be transferred to other languages or to a different setting [46].

As mentioned before, we cannot claim statistical representativeness as three languages are too small of a sample size and because we did not choose the languages at random. In the context of this case study, we argue that these languages are nevertheless best practice examples of delta languages and that even on these best practice examples we can yield the presented benefits.

During the evaluation, cases for different types of languages (architectural, behavioral, and structural), which had been developed under different circumstances, have been examined. The results are similar in all three cases and, therefore, we believe they would be similar in other cases, too. In addition, the language designers involved in the cases we use in our case study are either representatives of the delta modeling community or have been supervised by such.

Corresponding to Sect. 8.2, we assume, as long as the qualification of the language designers developing the manual written language and the language designers extending the generated language is on a similar level, the given results can be transferred.

9 Related work

In this section, we discuss related work in the area of variability modeling approaches, model transformation languages and approaches, which aim at deriving transformation languages for a specific base language.

9.1 Variability modeling

Approaches intending to model variability in modeling languages can be classified into three main directions [21,57]: annotative, compositional, and transformational variability modeling.

Annotative approaches consider one 150 % model representing all products of the product line. Variant annotations expressed using, e.g., UML stereotypes [13,66] or presence conditions [7] define which parts of the model have to be removed to derive a concrete product model. The orthogonal variability model (OVM) [37] captures the variability of product line artifacts in a separate variability model in which artifact dependencies serve as annotations. The variability modeling language [35] is a specialization of the OVM for architectural models. The XML-based Variant Configuration Language (XVCL) [26,48,64] is a frame-based [4] and language-independent annotative approach [6]. To facilitate reusing software assets among different product variants, software assets can be split into generic and adaptable fragments, called x-frames. Such x-frames can contain XVCL commands that, e.g., mark variation points. A XVCL processor traverses a hierarchy of x-frames and assembles the frames into one or more files. An enhanced version of XVCL, the Variant Configuration Language, is presented in [9] and a hybrid approach for feature oriented programming in XVCL is presented in [65]. Nevertheless, this approach is different from delta modeling since x-frames have to be designed especially for variability while

delta modeling is dedicated to vary an existing model after development. This, unfortunately, is quite often the case in industry, where a product line emerges from a single product.

Compositional approaches associate model fragments with product features that are composed for a particular feature configuration. In [24,36,57], aspect-oriented composition is used for constructing models. In [2], the composition of model fragments is performed by model superimposition. In feature-oriented model-driven development [53], a combination of feature-oriented programming and model-driven engineering, a product model is composed from a base module and a sequence of feature modules.

Transformational approaches express variability by transformation rules. The common variability language [23] provides means to express variability of a base model in a language that does not depend on the base modeling language. This is done by specifying rules that describe how model elements of a base model have to be substituted in order to obtain a particular product model. In [27], the model composition language MATA (modelling aspects using a transformation approach) is introduced, which enables the specification of variant features by graph transformation rules that modify kernel models. Graph transformation rules are also used in [51,60] to capture architectural variability. In [25], architectural variability is represented by change sets containing additions and removals of components and component connections that are applied to a base line architecture. Delta modeling also belongs to the group of transformational approaches.

Delta modeling has already been applied to several languages, such as the architectural description language MontiArc [20], Java in [43], Class Diagrams in [42] and Simulink models in [19]. In contrast to these publications, which illustrate applications of delta modeling to concrete languages, our work presents a method which allows synthesizing a delta language from the grammar of a given base language.

9.2 Model transformation languages

Delta languages can be classified as a special type of model transformation languages, in which delta models correspond to transformation rules. Out of the multitude of different model transformation approaches, graph-based transformation approaches [8] such as AGG [50], VIATRA [55], GReAT [1] or PROGRES [45] are essentially the most similar to delta modeling. Graph transformation rules usually consist of a LHS, a RHS, and often negative application conditions (NAC). The LHS describes the pattern to be searched for in the model to be transformed and the RHS describes the pattern which replaces the matched elements. A NAC represents a pattern that must not be found. In this way, powerful transformations can be formulated.

In the following, we outline the major differences between delta languages and typical graph-based transformation languages. These differences can as well be transferred easily to other types of model transformation languages.

D1 One difference is the way in which elements to be modified are specified. Delta models refer to concrete model elements, whereas graph transformation rules usually describe general patterns which can be mapped to a multitude of elements. Even though general patterns are a powerful feature, such generality is not needed and might even be inconvenient, if the intention is to modify one specific element. This inconvenience is caused by the necessity of additional application conditions to restrict the possible matches.

D2 A further difference concerns the need to specify NACs. In order to avoid that applying a transformation rule leads to an invalid model, the developer of a transformation rule has to specify NACs. For a transformation rule that adds a substate to a state, such a NAC can, e.g., express, that the state must not already contain a substate with the given name. Delta languages created according to our approach do not offer constructs to define NACs. This is done on purpose to simplify the specification of delta models as much as possible. Instead, we assume that these checks are implemented via context conditions that ensure that specific types of delta operations can(not) be applied. One such context condition can, e.g., ensure that adding a substate to a state is only possible if the state does not already contain a substate with the corresponding name. As presented in Sect. 4.3, some context conditions are generated and must, therefore, not be implemented by developers.

D3 Another difference is related to the modification operations that can be specified. A delta language provides a well-defined and restricted amount of delta operations that are used for model-specific modifications. In contrast to this, graph transformation rules are capable of modeling *arbitrary* modifications. Albeit such rules are more powerful, it is easier to specify and understand the restricted amount of delta operations offered by a delta language.

D4 The last difference concerns the syntax which transformation rules are based on. Most transformation languages solely operate on the abstract syntax of the models to be transformed [59]. The advantage of these approaches is that they can express transformations for *any* kind of model. However, the disadvantage is that the developer of a transformation rule inevitably needs to have a deep knowledge of the metamodel. In contrast to this, delta modeling allows reusing the concrete syntax of the corresponding modeling language.

9.3 Derivation of transformation languages

In [3], the metamodel of a pattern language, in which the LHS and RHS of a transformation rule are specified, is gen-

erated out of the metamodel of a modeling language. This derivation of the pattern language is particularly based upon a relaxation of the metamodel of the modeling language, as this metamodel cannot be used as a foundation for specifying patterns in transformation languages in general as a pattern might be an incomplete part of a valid model. After the pattern language metamodel has been generated, the user has to define the concrete syntax for the transformation language.

As Baar and Whittle [3] and Syriani et al. [33,49] semi-automatically generate pattern languages based on relaxation of the metamodels for describing the LHS and RHS of a transformation rule. Here, two slightly different metamodels addressing the requirements for a LHS/RHS pattern are derived.

In [14,15], a framework of a model-to-model transformation language is presented, which can be configured with different graphical source and target modeling languages [16]. In order to be able to use the concrete syntaxes of the modeling languages in transformation rules, they automatically generate a concrete syntax-based rule editor for the involved modeling languages. This generation step, however, requires the user to link the abstract syntax of the modeling languages to the concrete syntax. In this way, altogether a graphical model-to-model transformation language for two graphical base languages is derived semi-automatically.

In contrast to the approach presented in this paper, these approaches use a graphical notation for LHS and RHS patterns. Furthermore, these approaches require to constrain a pattern and pattern matching for the LHS, whereas our approach directly addresses the element to be transformed and instead of using two models to describe the LHS and RHS uses an integrated notation. Furthermore, our method clearly defines the necessary steps to derive a grammar for a textual delta language from a textual base language. This comprises both abstract and concrete syntax of the delta language.

A further type of approaches generates model transformation rules based on examples [28,54,61]. In such approaches, the user has to define how model elements of concrete source and target models should be interrelated. Afterwards, inferring transformation rules from these prototypical mappings is attempted. As it might not always be possible to propose the complete transformation rule, the user is able to refine the proposed transformation rules. In addition to that, approaches such as those presented in [5,47] assume that users demonstrate the effects of the desired model transformation by modifying a specific source model accordingly. This requires that modification operations performed by the user can be inferred or recorded. Based on these operations, constructing of a general transformation rule is attempted. Even though these works could also be used to derive transformation rules for delta operations, the user would need to define the exemplary mappings or to demonstrate the effects explicitly on his own. Furthermore, it will often be necessary to refine the proposed rules. In contrast to this, our approach derives suitable delta operations automatically based on the grammar of the language.

The most similar work to ours is the generation of a textual domain-specific transformation language (DSTL) for a textual base language described in [58]. A concrete DSTL for hierarchical automata, that could be created by this approach, is presented in [59]. In [58], the grammar for the DSTL is derived systematically from the grammar of the base language. The derivation is comparable to our approach since it also presents a systematic derivation of a delta language from the grammar of a base language. The major difference lies in the resulting languages. The differences *D2* and *D3* still hold between [58] and our approach since the applicability of transformation rules has to be restricted by NACs, and *all kinds* of model modifications and not only well-defined delta operations can be modeled. However, the differences *D1* and *D4* do not hold as the transformation rules in [58] also refer to concrete model elements and also reuse the concrete syntax of the corresponding modeling language.

10 Conclusion

Delta modeling is a modular, yet flexible approach to represent variability by explicitly capturing system changes. To use delta modeling for a certain modeling language that has no corresponding delta modeling language yet, a separate delta language has to be synthesized. Hence, to use delta modeling techniques within every step of a development process, one needs to create many delta languages manually.

To overcome this, we presented a method to synthesize delta languages from a textual base language definition. It decreases the effort of creating a delta language and allows to synthesize an initial delta language that then can be adapted and extended.

To evaluate our method, we used a comparative case study which compares existing originally handwritten delta languages to automatically generated ones and to the extended delta languages using well-defined metrics. Our evaluation has shown that the languages derived by the fully automatic approach are as semantically expressive as the handwritten ones.

We show that our method including the manual tailoring that is necessary to achieve the same syntactic expressiveness creates less effort than creating the handwritten language without our method. The tailored language is being able to process the same model as the handwritten language.

It furthermore creates a more consistent grammar. As the generator derives all languages in the same way, we assume it will be far easier to learn a new delta language, if one is familiar with the presented method and the base language. This will minimize training costs.

Our sample is too small for statistically representative results; our results are, therefore, limited to the cases we have studied.

If the MontiCore features used in our method also exist or may be simulated in other language development frameworks, our method can be applied to other frameworks as well. For a detailed description of these features we refer to [32].

Our experience as well as discussions with industrial users have shown that the delta language approach is of high value, as users trained in specific languages can more easily be accustomed to use a delta extension of that language and, therefore, still deal with the well-known concrete syntax.

References

1. Agrawal, A., Karsai, G., Shi, F.: Graph transformations on domain-specific models. Technical Report TN 37203, Institute for Software Integrated Systems, Vanderbilt University (2003)
2. Apel, S., Janda, F., Trujillo, S., Kästner, C.: Model superimposition in software product lines. In: International Conference on Model Transformation (ICMT) (2009)
3. Baar, T., Whittle, J.: On the usage of concrete syntax in model transformation rules. In: International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics (PSI) (2007)
4. Bassett, P.: Frame-based software engineering. Softw. IEEE **4**(4), 9–16 (1987)
5. Brosch, P., Langer, P., Seidl, M., Wimmer, M.: Towards end-user adaptable model versioning: the by-example operation recorder. In: International Workshop on Comparison and Versioning of Software Models (CVSM), pp. 55–60 (2009)
6. Clarke, D., Helvensteijn, M., Schaefer, I.: Abstract delta modeling. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE'10), pp. 13–22. ACM, New York (2010)
7. Czarnecki, K., Antkiewicz, M.: Mapping features to models: a template approach based on superimposed variants. In: International Conference on Generative Programming and Component Engineering (GPCE) (2005)
8. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–645 (2006)
9. Daniel, D., Jarzabek, S., Ferenc, R.: Configuring software for reuse with VCL. In: 13th Symposium on Programming Languages and Software Tools (SPLST), Szeged, pp. 16–30 (2013)
10. Feldt, R., Magizinius, A.: Validity threats in empirical software engineering research—an initial survey. In: Proceedings of the International Conference on Software Engineering and Knowledge Engineering, pp. 374–379 (2010)
11. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
12. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Future of Software Engineering 2007 at ICSE (FOSE) (2007)
13. Gomaa, H.: Designing Software Product Lines with UML. Addison Wesley Longman Publishing Co., Inc., Redwood City (2004)
14. Grønmo, R.: Using concrete syntax in graph-based model transformations. PhD thesis, University of Oslo (2009)
15. Grønmo, R., Møller-Pedersen, B.: Concrete syntax-based graph transformation. Research Report 389 (2009)
16. Grønmo, R., Møller-Pedersen, B., Olsen, G.K.: Comparison of three model transformation languages. In: European Conference on Model Driven Architecture—Foundations and Applications (ECMDA-FA), pp. 2–17 (2009)
17. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: MontiCore: a framework for the development of textual domain specific languages. In: International Conference on Software Engineering (ICSE) (2008)
18. Haber, A., Hölldobler, K., Kolassa, C., Look, M., Rumpe, B., Müller, K., Schaefer, I.: Engineering delta modeling languages. In: Proceedings of the 17th International Software Product Line Conference (SPLC'13), pp. 22–31. ACM, New York (2013)
19. Haber, A., Kolassa, C., Manhart, P., Nazari, P.M.S., Rumpe, B., Schaefer, I.: First-class variability modeling in matlab/simulink. In: International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS) (2013)
20. Haber, A., Kutz, T., Rendel, H., Rumpe, B., Schaefer, I.: Delta-oriented architectural variability using MontiCore. In: European Conference on Software Architecture (ECSA) (2011)
21. Haber, A., Rendel, H., Rumpe, B., Schaefer, I.: Delta modeling for software architectures. In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII, pp. 1–10. fortiss GmbH, Munich (2011)
22. Haber, A., Ringert, J.O., Rumpe, B.: MontiArc—architectural modeling of interactive distributed and cyber-physical systems. Technical Report AIB-2012-03, RWTH Aachen, Germany (2012)
23. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G., Svendsen, A.: Adding standardized variability to domain specific languages. In: International Software Product Line Conference (SPLC) (2008)
24. Heidenreich, F., Wende, C.: Bridging the gap between features and models. In: Aspect-Oriented Product Line Engineering (AOPLE) (2007)
25. Hendrickson, S.A., van der Hoek, A.: Modeling product line architectures through change sets and relationships. In: International Conference on Software Engineering (ICSE) (2007)
26. Jarzabek, S.: Variability management for product lines with XVCL. In: Proceedings of 11th International Conference on Software Product Lines (SPLC'07), Kyoto, September 10–14, 2007, Proceedings. Second Volume (Workshops), pp. 13–14 (2007)
27. Jayaraman, P.K., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model composition in product lines and feature interaction detection using critical pair analysis. In: International Conference on Model Driven Engineering Languages and Systems (MoDELS) (2007)
28. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model transformation by-example: a survey of the first wave. In: Düsterhöft, A., Klettke, M., Schewe, K.-D. (eds.) Conceptual Modelling and its Theoretical Foundations, pp. 197–215. Springer, Berlin (2012)
29. Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., Völkel, S.: Design guidelines for domain specific languages. In: OOPSLA Workshop on Domain-Specific Modeling (DSM) (2009)
30. Krahn, H.: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering. PhD thesis, RWTH Aachen University, Germany (2010)
31. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: modular development of textual domain specific languages. In: International Conference on Objects, Models, Components, Patterns (Tools) (2008)
32. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. Int. J. Softw. Tools Technol. Transf. (STTT) **12**(5), 353–372 (2010)
33. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Explicit transformation modeling. In: International Conference on Models in Software Engineering (MoDELS) (2010)

34. Look, M., Navarro Perez, A., Ringert, J.O., Rumpe, B., Wortmann, A.: Black-box integration of heterogeneous modeling languages for cyber-physical systems. In: GEMOC Workshop 2013—International Workshop on the Globalization of Modeling Languages, Miami (2013)
35. Loughran, N., Sánchez, P., Garcia, A., Fuentes, L.: Language support for managing variability in architectural models. In: Pautasso, C., Tanter, É. (eds.) Software Composition. Lecture Notes in Computer Science (LNCS). Springer, Berlin (2008)
36. Noda, N., Kishi, T.: Aspect-oriented modeling for variability management. In: International Software Product Line Conference (SPLC) (2008)
37. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering—Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
38. Purdom, P.: A sentence generator for testing parsers. BIT Numer. Math. **12**(3), 366–375 (1972)
39. Rumpe, B.: Modellierung mit UML, 2nd edn. Xpert.press. Springer, Berlin (2011)
40. Rumpe, B.: Agile Modellierung mit UML, 2nd edn. Springer, Berlin (2012)
41. Runeson, P., Host, M., Rainer, A., Regnell, B.: Case Study Research in Software Engineering: Guidelines and Examples. Wiley Blackwell, New York (2012)
42. Schaefer, I.: Variability modelling for model-driven development of software product lines. In: International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS) (2010)
43. Schaefer, I., Bettini, L., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: International Conference on Software product lines (SPLC) (2010)
44. Schindler, M.: Eine Werkzeuginfrastruktur zur Agilen Entwicklung mit der UML/P. Aachener Informatik Berichte, Software Engineering. Shaker Verlag, Aachen (2012)
45. Schürr, A.: Progres: A vhl-language based on graph grammars. In: International Workshop on Graph-Grammars and Their Application to Computer Science (1990)
46. Shadish, W.R., Cook, T.D., Campbell, D.T.: Experimental and Quasi-Experimental Designs for Generalized Causal Inference, 2nd edn. Houghton Mifflin, Boston (2001)
47. Sun, Y., White, J., Gray, J.: Model transformation by demonstration. In: Schürr, A., Selic, B. (eds.) Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science (LNCS), pp. 712–726. Springer, Berlin (2009)
48. Swe, S.M., Zhang, H., Jarzabek, S.: XVCL: a tutorial. In: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE), pp. 341–349. ACM, New York (2002)
49. Syriani, E., Gray, J., Vangheluwe, H.: Modeling a model transformation language. In: Reinhartz-Berger, I., Sturm, A., Clark, T., Cohen, S., Bettin, J. (eds.) Domain Engineering, pp. 211–237. Springer, Berlin(2013)
50. Taentzer, G.: Agg: a graph transformation environment for modeling and validation of software. In: International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE) (2003)
51. Tamzalit, D., Mens, T.: Guiding architectural restructuring through architectural styles. In: International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS) (2010)
52. Trochim, W., Donnelly, J.: The Research Methods Knowledge Base. Atomic Dog Publishing, Cincinnati (2006)
53. Trujillo, S., Batory, D., Díaz, O.: Feature oriented model driven development: a case study for portlets. In: International Conference on Software Engineering (ICSE) (2007)
54. Varró, D.: Model transformation by example. In: International Conference on Model Driven Engineering Languages and Systems (MoDELS), pp. 410–424 (2006)
55. Varró, D., Balogh, A.: The model transformation language of the viatra2 framework. Sci. Comput. Program. **68**(3), 187–207 (2007)
56. Völkel, S.: Kompositionale Entwicklung domänenspezifischer Sprachen. PhD thesis, TU Braunschweig, Germany (2011)
57. Völter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software development. In: International Conference on Software product lines (SPLC) (2007)
58. Weisemöller, I.: Generierung domänenspezifischer Transformationssprachen. Aachener Informatik Berichte, Software Engineering. Shaker Verlag, Maastricht (2012)
59. Weisemöller, I., Rumpe, B.: A Domain Specific Transformation Language. In: Workshop on Models and Evolution (ME) (2011)
60. Wermelinger, M., Fiadeiro, J.L.: A graph transformation approach to software architecture reconfiguration. Sci. Comput. Program. **44**(2), 133–155 (2002)
61. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards model transformation generation by-example. In: Hawaii International Conference on System Sciences (HICSS) (2007)
62. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Norwell (2000)
63. XText website. http://www.eclipse.org/Xtext/
64. Zhang, H., Jarzabek, S.: XVCL: A Mechanism for Handling Variants in Software Product Lines. Sci. Comput. Program. **53**(3), 381–407 (2004)
65. Zhang, H., Jarzabek, S.: A Hybrid approach to feature-oriented programming in XVCL. In: Prpceedings of 14th International Conference on Software Product Lines: Going Beyond (SPLC'10), Jeju Island, pp. 440–445 (2010)
66. Ziadi, T., Hélouët, L., Jézéquel, J.-M.: Towards a UML profile for software product lines. In: Workshop on Product Familiy Engineering (PFE) (2003)