DeltaEcore—A Model-Based Delta Language Generation Framework

Christoph Seidl[1], Ina Schaefer[2], Uwe Aßmann[1]

[1]Software Technology Group
Technische Universität Dresden
christoph.seidl@tu-dresden.de, uwe.assmann@tu-dresden.de

[2]Software Engineering Institute
Technische Universität Braunschweig
i.schaefer@tu-bs.de

Abstract: Software product lines (SPLs) and software ecosystems (SECOs) represent families of closely related software systems in terms of configurable variable assets. Delta modeling is an approach for capturing variability resulting from different configurations and for deriving concrete software products of an SPL or SECO through transformation. Even though the general concepts of delta modeling are language-independent, custom delta languages are required for all source languages, which are tedious to create and lack interoperability due to different implementation technologies. In this paper, we present a framework to automatically derive delta languages for textual or graphical languages given as EMOF-based meta models. We further illustrate how to automatically generate the syntax and large parts of the semantics of the derived delta language by inspecting the source language's meta model. We demonstrate our approach by applying our implementation DeltaEcore to four selected source languages.

1 Introduction

Software product lines (SPLs) [PBvdL05] and software ecosystems (SECOs) [Bos09] are approaches to reuse in the large where families of closely related software systems are modeled in terms of configurable functionality often referred to as *features*. An SPL has a *closed variant space* where the set of all possible features is explicitly known making it (theoretically) possible to determine all valid *variants* of the SPL a priori [PBvdL05]. In contrast, SECOs have an *open variant space* [Bos09] where not necessarily all features are known by a central instance at any particular time. A *configuration* for one member of the software family is represented by a valid subset of all possible features. In order to derive a concrete software system for a configuration, a variability mechanism has to build a variant from all realization parts related to the features present in the configuration. As most software systems consist of multiple artifacts for different purposes (e.g., design models, source code, configuration files, documentation material etc.), a variety of languages has to be made subject to variability in SPLs and SECOs. With suitable meta models for the

respective languages, all these artifacts can uniformely be regarded as models allowing to handle textual as well as graphical languages.

In our work, we chose the transformational variability mechanism *delta modeling* [SBB+10] to represent variability due to its ability to handle both SPLs and SECOs as well as configuration and evolution (see Section 2). Delta modeling alters a given base variant of an SPL or SECO by adding, modifying and removing parts to transform the system into a variant conforming to the provided configuration. In delta modeling, transformation steps are described in a domain-specific language-dependent *delta language*, which restricts transformation operations and which is closely tied to its source language, e.g., Delta Java [SBB+10] as delta modeling language for Java.

With multiple different languages specifying a family of software systems (e.g., design models, source code etc.), a variability mechanism needs to be applicable to all languages whose artifacts are affected by different configurations. For delta modeling, this means that all languages and their meta models need to have a respective delta language to alter them programmatically. This is complex as a) many languages, in particular domain-specific languages, do not have a pre-defined delta language and b) new languages may be introduced and existing ones may be altered as part of system evolution requiring adaptation of the respective delta language as well. Creating delta languages manually requires extensive efforts and delta languages created by different developers often lack interoperability due to different implementation technologies.

In our approach, we address the problems arising from manually creating delta languages. We introduce a model-based framework to define delta languages for source languages with an EMOF[1]-based meta model. We further define six types of standard delta operations and illustrate how to analyze a source language's meta model to derive large parts of the delta operations for a suitable delta language. We generate syntax, semantics and tooling for these delta languages including editor support, parsers and interpreters. The generated delta languages seamlessly integrate into a common variant derivation mechanism so that they can be used to create variants of an SPL or SECO and are fully interoperable with other delta languages created with this framework.

This paper is structured as follows: Section 2 introduces delta modeling with its benefits and limitations as well as a running example used throughout the paper. Section 3 explains our delta language generation framework and illustrates how to derive suitable delta operations from analyzing a source language's meta model. Section 4 demonstrates the implementation of these concepts in our tool DeltaEcore. Section 5 shows the feasiblity of our approach by selected case studies before Section 6 discusses related work and Section 7 closes with an outlook to future work.

[1] EMOF (Essential MOF) is a subset of the Meta-Object Facility (MOF) 2.0 standard for model-driven engineering by the Object Management Group (OMG), see `http://omg.org/mof`

2 Delta Modeling

Delta modeling is an approach for capturing variability in software families and for deriving individual products [SBB+10]. The general idea is to transform one valid variant of the family into another variant realizing a different valid set of features by means of adding, modifying or removing elements of the first variant. Within the approach, a *delta module* is used to bundle the transformation operations associated with (part of) a particular configurable unit of functionality or combinations thereof. The individual transformations in a delta module are performed by application of *delta operations*, which are custom-defined transformation procedures specified individually for each language. Within this paper, we use the term *source language* for the original language (e.g., Java) and *delta language* for the language in which delta modules containing delta operations are specified (e.g., Delta Java [SBB+10]). A source language in delta modeling may be textual, graphical or in any other representation. Delta languages are usually specified textually [SBB+10, DS11], but there also are attempts to specify them graphically [HKM+13]. To derive a particular product in delta modeling, a set of delta modules is brought into a suitable order and applied by executing the respective delta operations sequentially.

To illustrate the concepts in this paper, we use the example of Software Fault Trees (SFTs) applied in safety-critical software to successively decompose a root fault into logical combinations of its constituent faults in order to determine causes for the root fault's appearance [Lev95]. An SFT is a tree consisting of gates representing logical and/or operations as well as intermediate faults, which are refined further, and basic faults, which are considered atomic. Basic faults are assigned an individual probability of occurrence, which can be used to derive metrics for the likelihood of more complex faults activating. Figure 1a) shows an example SFT.

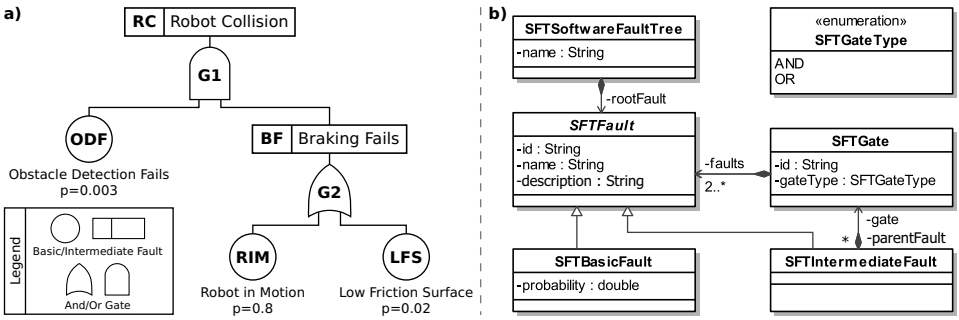

Figure 1: a) Example SFT. b) Meta model for SFTs.

When safety-critical software systems are created from a software family in the sense of an SPL or SECO, the respective safety artifacts describing the system for analysis and certification need to be altered equivalently to the system itself for different configurations [SSA13, DL04]. Thus, when using delta modeling, languages such as SFTs need a delta language to express variability. We chose SFTs as a running example as they demonstrate many of the principal challenges in creating custom delta languages yet are sufficiently

comprehensible. Figure 1b) shows the meta model for SFTs we use throughout the paper. The meta class `SFTSoftwareFaultTree` represents the root element of the SFT. In addition, `SFTFault` is the abstract base class for its specializations `SFTBasicFault` and `SFTIntermediateFault` representing the respective faults. Finally, `SFTGate` represents logical gates with the respective logical operator of the enumeration `SFTGateType`. In the meta model, we distinguish structural features of meta classes into references relating elements to instances of meta classes and attributes having values with basic types, custom data types or enumerations. Furthermore, we distinguish single-valued references having an upper bound of one and many-valued references having an upper bound greater than one resulting in a (possibly ordered) set of values.

```
 1 delta "RefineObstacleDetection"
 2   dialect <http://vicci.eu/ecosystem/sft/1.0>
 3   requires <../core/RobotCollision.sft>
 4 {
 5   removeFaultFromFaultsOfGate(<ODF>, <G1>);
 6   SFTIntermediateFault odf = new SFTIntermediateFault(id: "ODF",
 7     name: "Obstacle Detection Fails");
 8   addFaultToFaultsOfGate(odf, <G1>);
 9
10   SFTGate g3 = new SFTGate(id: "G3", gateType: SFTGateType.AND);
11   setGateOfIntermediateFault(g3, odf);
12
13   addFaultToFaultsOfGate(new SFTBasicFault(id: "BSF",
14     name: "Bump Sensor Fails", probability: 0.003), g3);
15   addFaultToFaultsOfGate(new SFTBasicFault(id: "DSF",
16     name: "Distance Sensor Fails", probability: 0.0007), g3);
17 }
```

Listing 1: Example usage of DeltaSFT to alter SFTs in the course of variability.

In general, a delta language should provide operations to create new instances of all concrete meta classes and to reference existing elements. *DeltaSFT* as delta language for SFTs conforming to the presented meta model should further allow to add and remove faults to/from the many-valued `faults` reference of gates as well as to set and unset the value of the single-valued `gate` reference of intermediate faults. Furthermore, DeltaSFT has to support modification of the attribute `name` for both fault trees and faults as well as `probability` of basic faults and `gateType` of gates by assigning a new value. The `id` of both faults and gates is closely related to the identity of the respective elements and, thus, should not be subject to changes due to variability. An example of a delta module in DeltaSFT is provided in Listing 1. It modifies an SFT capturing the causes for the collision of a domestic robot. The basic variant of the SFT is loaded in l. 3 and modified in ll. 5–16 to include fault propagation paths for an add-on distance sensor by applying delta operations specific to the source language of SFTs.

The general concepts of delta modeling can be seen as a specialized form of model transformation [MVG06]. In contrast to a general model transformation engine, a delta language only provides selected modification operations required for expressing variability. Operations that should not be performed as part of variability, such as changing IDs, are explicitly prohibited by not providing the respective delta operations. Furthermore, operations may be specified to respect the syntactical and semantical constraints of the source language, e.g., by avoiding dangling references. Finally, variability engineers are not required to learn

or understand the full scope of a general model transformation engine but only that of the reduced functionality of the delta language.

Delta modeling has multiple beneficial qualities when used as variability mechanism. For one, it can handle configuration (variability in space) as well as evolution (variablity in time) within a single notation [SBB+10, DS11] allowing both to derive products and to modify the SPL or SECO in response to changed or new requirements. Furthermore, delta modeling does not depend on a closed variant space as in SPLs but can deal with an open variant space where not necessarily all configuration options are known in advance as found in SECOs [Bos09, SA13], which is a discriminating difference to annotational variability mechanisms [SRC+12] often used with SPLs.

These characteristics and the fact that the general concepts of delta modeling are language independent make it a very suitable option for SPL and SECO development. However, for a practical application of delta modeling to a particular language or meta model, an implementation of a custom delta language for its source language is required. Furthermore, a variability modeling approach based on delta modeling is only applicable if all languages of the SPL or SECO that are affected by variability support it. Even though implementations of delta languages exist for source languages such as Java [SBB+10], Class Diagrams [Sch10], Matlab/Simulink [HKM+13] or Component Fault Diagrams (CFDs) [SSA13], they are currently incompatible with one another and less known languages need individual implementation of a delta language.

Creating a delta language manually for a specific source language or meta model is tedious as not only the language's syntax and semantics have to be devised but also the tooling to create delta modules and derive product variants needs to be created. This results in a number of problems: First, most languages do not possess a delta language as it would have to be defined manually. Second, implementations lack robustness as reuse of common technologies is not possible. Third, delta languages created by different developers lack interoperability so that multiple tools are required to handle variability of different source languages. Automatically generating delta languages on basis of a common framework may address these problems. However, existing approaches [HHK+13] are limited to deriving the syntax of delta languages for textual languages from grammars and cannot generate their semantics or tooling for product derivation.

3 Delta Language Generation Framework

In this paper, we present a framework to create custom delta languages for source languages given as EMOF-based meta models. Within our framework, we use information from analyzing a source languages's meta model to derive syntax and large parts of the semantics for the model representation of a delta language with a concrete textual syntax. For this purpose, we use two languages represented by meta models with concrete textual syntax: 1) The *common base delta language*, which provides functionality common to all delta languages such as creating and referencing elements and 2) a *delta dialect*, which provides delta operations specific to the source language. A delta language is created by combining

the common base delta language with a delta dialect specific to the respective source language. This general architecture is illustrated in Figure 2.

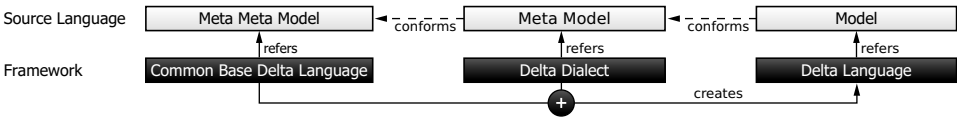

Figure 2: Architecture of the delta language generation framework.

3.1 Common Base Delta Language

The common base delta language operates on the level of the meta meta model (EMOF) using e.g., `EReferences` as elements, but not their instances in the meta model of the source language, such as the reference `faults` of the meta class `SFTGate` in the meta model for SFTs defined in Figure 1b). Hence, the common base delta language requires no knowledge of the source language's meta model so that it is provided entirely by the framework. The common base delta language represents the skeleton of the custom delta language that is to be created. Constructs defined by the common base delta language include a) references to other delta modules or models (e.g., requires), b) dynamically created constructors with named parameters to instantiate meta classes, c) references to existing model elements (language dependent identifiers are possible), d) definition of variables and constants and e) invokation of delta operations with arguments.

These constructs are available in all delta languages created using the framework, but can be defined independently from the concrete source language. In order to avoid having to define them for each delta language individually, we provide these constructs as part of the framework and share them between different delta languages. As the common base delta language is defined in a meta model, we are able to perform operations such as type checks to ensure that the types of referenced objects, variables and parameters are compatible. We further provide a concrete textual syntax with the meta model, which is used as basis for the textual custom delta language when combining the common base delta language with a delta dialect.

3.2 Delta Dialect

A delta dialect defines delta operations suitable to expressing variability for a particular source language, e.g., to add faults as children of a gate for SFTs. Thus, a delta dialect is the part of a custom delta language that ties to the meta model of a specific source language. The delta language itself is created by combining the common base delta language with the respective delta dialect for the source language. We specify the structure for delta dialects using a meta model and further provide a concrete textual syntax (see Listing 2 in

Section 4). The custom delta language is created by dynamically introducing references between the meta models of the common base delta language and the respective delta dialect. Along with the resulting meta model, we provide a concrete textual syntax for the resulting custom delta language that is synthesized from the textual syntax of the common base delta language and the meta classes in the source language (see Listing 1 in Section 2). Hence, delta modules may be specified textually and principally also in other forms, e.g., graphically.

3.3 Delta Operations

A delta dialect is specified for a particular source language by the users of our framework by defining suitable delta operations for the source language. In our running example, we illustrated the need for five types of operations: setting und unsetting the value of single-valued references, adding and removing values of many-valued references and modifying the value of attributes. We further identified the need for a sixth operation that can insert a value into a many-valued reference at a specified position provided that the set of values is ordered. Using these six types of operations, we define semantics for standard delta operations used for variability modeling with EMOF-based models and illustrate how to derive them from a source notation's meta model. Furthermore, we also support developers in creating custom delta operations with user-defined semantics to realize domain-specific operations.

Set/Unset Delta Operations are used to alter the value of a single-valued reference. A set delta operation assigns a new value to a specified single-valued reference, whereas an unset delta operation replaces the current value with the default value for that reference as defined in the meta model.

We derive set and unset operations from a source language's meta model by collecting all references in a set that are changeable and single-valued. For each reference in the set, we define both a set and unset delta operation. The delta dialect for our running example in Listing 2 in Section 4 contains definitions for two set delta operations (ll. 7/8, 20/21) and two unset delta operations (ll. 9/10, 22).

Add/Insert/Remove Delta Operations are provided to manipulate the set of values of many-valued references. An add operation appends a given element to the set of values and a remove operation detaches it from the set. Thus, the semantics of a remove operation is different from that in other approaches to delta modeling [SBB$^+$10, DS11] where it completely erases an element from the model whereas, in our case, the element is only detached from the specified list of references. An insert operation places the element at a certain position within the set of values, which is only sensible if the set is ordered.

We derive add, insert and remove delta operations in a similar way to set and unset delta operations: We first collect a set of all references that are changeable and, in this case, many-valued. As insert delta operations are only sensible for ordered sets of values, we further exclude references that are marked as being unordered for this type of operation. For each reference in the set, we create the respective delta operations. The delta dialect for

our running example in Listing 2 contains one add delta operation (l. 28) and one remove delta operation (ll. 29/30). As none of the many-valued references of the meta model is marked as being ordered, no insert delta operations are required.

Modify Delta Operations are used to alter the values of an attribute. In contrast to manipulating referenced values, modification of attribute values is free of side effects (e.g., automatically updated opposite references). Hence, we decided that users of a delta language should be made aware of this difference so that we distinguish set and modify delta operations. In consequence, modify delta operations have a different meaning from that in other approaches to delta modeling [SBB+10, DS11] where they are used solely to signal that the contents of a hierarchically decomposed element are being altered. We do not require such a marker as we can use references to target elements directly even if they are nested within a containment hierarchy.

We derive modify delta operations from the provided meta model by inspecting all of its concrete (i.e., non-abstract) meta classes. For each of these meta classes, we iterate over the attributes and collect those that are changeable and not marked as ID. We decided not to allow modification of IDs by default as an *identifier* is tightly connected to the *identity* of an element and, thus, should not be changed as part of variability modeling. Instead, the element itself should be replaced. For each attribute in this set, we then generate a modify delta operation. The delta dialect for our running example in Listing 2 defines seven modify delta operations (ll. 11–18, 23–26, 31).

Custom Delta Operations are used to declare delta operations with user-defined domain-specific semantics that could not be expressed using the generated delta operations. This enables creators of a delta language to utilize knowledge of the semantics of the source language to provide specifically tailored operations, e.g., to avoid dangling references according to the constraints of the source language. As the semantics of these operations depends entirely on the behavior intended by the creator of the delta language, the implementation to interpret the respective custom delta operations needs to be provided manually.

We explicitly decided to not include two specific operations in the set of standard delta operations that may have to be realized as custom delta operations: For one, we refrained from defining a replace delta operation as it inherently depends on the semantics of the source language whether elements of the exact same type, those compatible in the sense of subtype polymorphism or semantically equivalent elements may be used as substitutes. Furthermore, we did not define a standard delete delta operation that completely erases an element from the model along with all its references as this operation would have too many (potentially unintended) side effects to be sensible for variability modeling in general. Hence, the element either has to be deleted step by step using standard delta operations or a custom delta operation specific to the source language has to be defined, which may be done for abstract meta classes to cover multiple concrete meta classes at once if no fine-grained control is required.

4 Implementation

We have realized the concepts presented in this paper using Ecore from the Eclipse Modeling Framework[2] (EMF) as meta modeling notation supporting EMOF. Our implementation is called *DeltaEcore* and is available for download at `http://deltaecore.org`. A variety of tools exists for Ecore to create model representations of both textual and graphical languages allowing DeltaEcore to target a wide range of source languages.

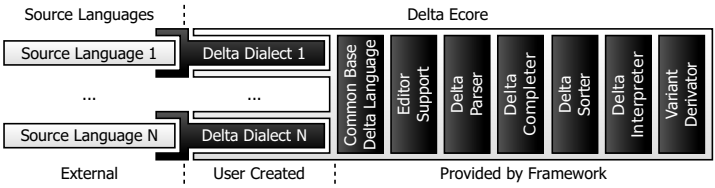

Figure 3: Implementation components of DeltaEcore.

Figure 3 illustrates the main implementation components of DeltaEcore, which we reference in the following using an italic type. Along with the *Common Base Delta Language*, DeltaEcore provides *Editor Support* for the derived delta languages including syntax highlighting, auto completion etc. as well as a *Delta Parser* to create a model representation from the textual syntax of a delta language. Each delta module explicitly specifies which models it alters and which other delta modules it depends on (if any) through a requires relation. When applying a set of delta modules, the *Delta Completer* collects all models to be altered and all (transitively) required delta modules, the *Delta Sorter* performs topological sorting to establish a suitable application order for the delta modules, the *Delta Interpreter* applies delta modules and their delta operations with the help of the generated delta dialect specific interpreters and the *Variant Derivator* assembles all affected models to store them as variant of the SPL or SECO. All these components are provided by DeltaEcore so that merely a *Delta Dialect* for a *Source Language* has to be defined by users of the framework in order to create a delta language. We use the steps described in Section 3.3 to automatically generate standard delta operations for a specified source language. In our implementation, we generate a model representation of these delta operations enabling us to enforce type safety when combining a delta dialect with the common base delta language.

Listing 2 shows the textual representation of a delta dialect for SFTs conforming to the meta model introduced in Figure 1b) as generated by DeltaEcore. When combining this delta dialect with the common base delta language, DeltaSFT is created, which can be used to specify variability for SFTs in delta modules such as the one depicted in Listing 1 of Section 2. In the `configuration` section of the delta dialect, the meta model of the source language is identified by specifying its URI as parameter to the `metaModel` key (l. 3). Furthermore, it is possible to optionally provide a custom `identifierResolver`—a Java class used to resolve references to elements within the meta model (l. 4). The default implementation uses attributes flagged as ID in Ecore to resolve references. However, it may be necessary to use custom identifiers such as with hierarchically structured models without

[2] `http://eclipse.org/modeling/emf`

```
1 deltaDialect {
2   configuration:
3     metaModel: <http://vicci.eu/ecosystem/sft/1.0>;
4     identifierResolver: eu.vicci.ecosystem.sft.delta.SFTIdentifierResolver;
5
6   deltaOperations:
7     setOperation setRootFaultOfSoftwareFaultTree(SFTFault value,
8       SFTSoftwareFaultTree[rootFault] element);
9     unsetOperation unsetRootFaultOfSoftwareFaultTree(
10      SFTSoftwareFaultTree[rootFault] element);
11    modifyOperation modifyNameOfSoftwareFaultTree(String value,
12      SFTSoftwareFaultTree[name] element);
13
14    modifyOperation modifyNameOfBasicFault(String value, SFTBasicFault[name] element);
15    modifyOperation modifyDescriptionOfBasicFault(String value,
16      SFTBasicFault[description] element);
17    modifyOperation modifyProbabilityOfBasicFault(Double value,
18      SFTBasicFault[probability] element);
19
20    setOperation setGateOfIntermediateFault(SFTGate value,
21      SFTIntermediateFault[gate] element);
22    unsetOperation unsetGateOfIntermediateFault(SFTIntermediateFault[gate] element);
23    modifyOperation modifyNameOfIntermediateFault(String value,
24      SFTIntermediateFault[name] element);
25    modifyOperation modifyDescriptionOfIntermediateFault(String value,
26      SFTIntermediateFault[description] element);
27
28    addOperation addFaultToFaultsOfGate(SFTFault value, SFTGate[faults] element);
29    removeOperation removeFaultFromFaultsOfGate(SFTFault value,
30      SFTGate[faults] element);
31    modifyOperation modifyGateTypeOfGate(SFTGateType value, SFTGate[gateType] element);
32 }
```

Listing 2: Textual representation of a delta dialect for SFTs.

unique identifiers. The characteristics of the identifiers depend on the source language so that the implementation of the respective identifier resolver is delegated to the creator of the delta language if the standard behavior does not suffice.

In the `deltaOperations` section (ll. 6–31), signatures for the delta operations provided within the custom delta language are given. All six types of standard delta operations are supported using distinct keywords for the different types of operations (e.g., `setOperation` or `addOperation`). In addition, it is possible to specify custom delta operations using the keyword `customOperation`, which may have arbitrary parameters and require a manual implementation of their semantics. In Listing 1, DeltaSFT is created by combining the common base delta language with the delta dialect of Listing 2 using the `dialect` keyword in l. 2.

When deriving standard delta operations, we synthesize names for the derived delta operations from the meta model of the source language to provide a naming convention, e.g., `setRootFaultOfSoftwareFaultTree` in Listing 2. However, these names may be changed at will by creators of delta dialects. To guide the process of deriving delta operations, we provide a graphical user interface allowing the deselection of undesired standard delta operations before generation. For all delta operations defined in a delta dialect, implementation classes for an interpreter of the custom delta language are generated. With the defined semantics of standard delta operations, it is possible to completely generate

the implementation for set/unset, add/insert/remove and modify delta operations. The semantics of custom delta operations is not formally defined and, thus, their interpretation needs to be implemented manually.

5 Case Study

In our case study, we evaluate the suitability of the concepts presented in this paper on four different languages using our tool DeltaEcore: Software Fault Trees [Lev95] (SFTs), Component Fault Diagrams [SSA13, KLM03] (CFDs), Checklists [Lev95] (CLs) and the Goal Structuring Notation [KW04] (GSN). An example of SFTs was already presented in Figure 1 and examples of CFDs, CLs and the GSN can be found in Figure 4. All these languages stem from the area of certifying safety-critical systems, but they contain many different features representing a wide range of languages. The abstract syntax of SFTs is represented by a tree, that of CFD and CLs by a reducible graph and that of GSN by a general graph. SFTs, CFDs and GSN have a graphical syntax, whereas CLs have a textual syntax. Finally, the GSN may reference model elements from SFTs, CFDs and CLs interconnecting the languages.

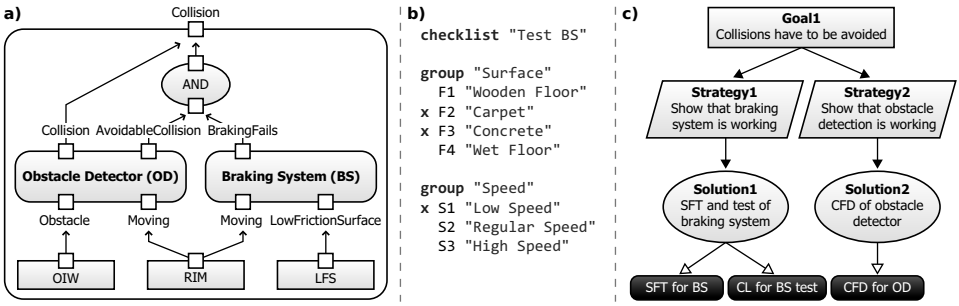

Figure 4: Example of languages used in the case study: a) CFDs, b) CLs, c) GSN.

In particular, we consider three research questions: *RQ1*: Is it possible to generate custom delta languages that are expressive enough to handle the required variability of the source language? *RQ2*: Is our approach capable of dealing with scenarios where other than the derived delta operations are required? *RQ3*: Are the generated methods sound (i.e., useful, fit for purpose and non-redundant)?

For CFDs, a delta language was already presented as part of our previous work [SSA13]. For SFTs, CLs and GSN, we manually created the respective delta languages to have a reference for comparison with delta languages generated by DeltaEcore. To answer our research questions, we inspected the delta operations derived from the languages' meta models and analyzed how complex the creation of custom delta operations and their implementations is in terms of lines of code (LOCs).

We created delta languages for these source languages using DeltaEcore. In Table 1, we provide metrics for the generated languages. The column "Generated" contains the

number of all generated delta operations, “Excess” counts those delta operations that are redundant (e.g., providing access to an opposite reference), “Not Ideal” lists the number of operations that were perceived as not being elegant for the intended purpose (e.g., setting the bounding box for the graphical representation of an element instead of moving and resizing it) and “Restrict” states the number of generated methods that had to be removed in order to disallow access to model elements that should not be affected by variability modeling. Finally, “Custom” lists the number of custom delta operations used in the delta language and “LOC” states the number of lines of code required to implement their intended semantics.

Source Language	Generated	Excess	Not Ideal	Restrict	Custom	LOC
SFT	15	2	0	0	0	0
CFD	39	12	2	17	6	31
CL	10	0	0	0	0	0
GSN	26	16	3	1	4	33

Table 1: Results of deriving delta dialects for the source languages of the case study.

The generated standard delta operations of all delta dialects were sufficient to handle variability in the respective source languages with regard to our original expectations. However, CFDs and GSN have a relatively large number of excess methods. This is mostly due to the presence of multiple opposite references where delta operations were generated for both the original and opposite reference creating redundancy. Furthermore, a large number of delta operations of the delta dialect for CFDs had to be removed in order to restrict access similarly to the original delta language for CFDs. However, we consider 13 of these 17 delta operations as being useful and merely had not included them in the original delta language due to the implementation effort at the time. To realize additional delta operations, CFDs required six and GSN four custom delta operations with 31 and 33 LOC respectively.

With regard to our research questions, we come to the following conclusions: Using DeltaEcore, it was possible to completely generate delta languages for the respective source languages that are expressive enough to handle variability resulting in a positive answer to *RQ1*. Even though it was possible to alter all elements with the derived delta operations, in some cases, providing more elegant delta operations was desirable. For example, the generated delta operations to alter the visual appearance of CFD elements suggested setting the bounding box of the element whereas the source language used delta operations to move and resize the element, which seemed more intuitive to use. Delta operations missing from the generated delta dialect could be realized by custom delta operations and manual implementation of the semantics in the dialect interpreter. Furthermore, access to elements considered immutable in the course of variability could be restricted by omitting the respective delta operations resulting in a positive answer to *RQ2*. The relatively large number of excess methods creates redundancy so that not all of the generated delta operations are considered sound with respect to our research questions resulting in a negative answer to *RQ3*. We will inspect how to reduce the number of redundant methods especially with regard to opposite references.

Threats to validity of our case study mainly come from selection of the source languages. Even though we used languages representing different characteristics, all four inspected source languages stem from the same domain of safety-critical systems so that they may not necessarily be representative for languages of other domains. Furthermore, the meta models for all source languages were created by the authors of this paper and, thus, may reflect a certain style of modeling. Finally, the inspected meta models are relatively small in comparison to those for languages such as Java.

6 Related Work

Multiple publications exist that present individual delta languages for particular source languages, such as for Java [SBB+10], Class Diagrams [Sch10], State Charts [LSKL12], Component Fault Diagrams [SSA13], the architectural language MontiArc [HKR+11] or Matlab/Simulink [HKM+13]. However, these delta languages are tightly integrated with their source languages and, thus, serve as archtypes of syntax and semantics of delta languages, but not as basis for generating custom delta languages for arbitrary meta models.

The work related closest to ours is that of Haber et al. [HHK+13] as it has the similar goal to generate a delta language for a given source language. They derive the concrete syntax for a custom delta language from a provided textual source language given as grammar by means of grammar extension. In contrast, we analyze the source language's abstract syntax to generate a delta language external to the source language. We use a similar concept of a common delta language. However, our common base delta language is represented as a meta model, which allows operations such as type checking whereas their common delta language merely consists of a grammar. In addition, their approach is limited to textual source languages whereas ours targets meta models and, thus, can create delta languages for models in textual, graphical or any other representation. Furthermore, their approach only generates the syntax of a delta language whereas ours generates large parts of an interpreter and an integration into a common variant derivation mechanism as well.

Another approach closely related to ours is that of Sánchez et al. [SLFG09] where a framework may be used to define domain-specific languages for variability management in a particular target meta model. In the extension of the work by Zschaler et al. [ZSS+10], SPL technologies are bootstrapped to create a family of these languages. Similar to our approach, the authors define modification operations external to the target meta model. However, they do not provide defined semantics for standard operations, but have language creators implement each operation using a general purpose model transformation language.

FeatureHouse [AKL13] is an approach for generalizing software composition by superimposition for artifacts written in different languages. FeatureHouse can be seen as a language workbench for feature-oriented variability modeling languages, which is similar to our approach for delta-oriented variability modeling. However, FeatureHouse does not operate on meta models of the source languages, but relies on the parse tree for the considered language and the concept of feature structure trees (FSTs), which resemble abstract syntax trees. The FSTs can be composed using a set of predefined operations with associated

semantics similar to the standard delta operations we provide. So far, FeatureHouse was only used for textual languages while our approach is more generally applicable for textual as well as for graphical languages.

The Common Variability Language (CVL) as a standardization effort for variability languages is closely related to our approach in that it has the goal to extend arbitrary MOF-based models with a variability mechanism. CVL defines semantics of certain standard operations that may be performed as part of variability modeling similar to our approach. However, CVL utilizes an annotational variability mechanism that depends on a closed variant space and, thus, may not be used with SECOs.

Besides approaches providing or generating languages to specifically handle variability, there are also more general approaches to model transformation that can be utilized for similar purposes. Rumpe and Weisemöller [RW11] generate a domain specific model transformation language from the concrete syntax of a source language. However, their focus is not on variability so that they do not provide standard variational operations with defined semantics or a variant derivation mechanism.

In addition, there are multiple general purpose model transformation approaches of which graph-based approaches are most suitable for variability modeling [CH06] with specifications such as QVT[3] and languages targeting Ecore such as ATL[4] or ETL[5]. However, the use of general purpose model transformation engines to express variability is problematic. Such languages are not tailored to the field of variability management with the result that they may be too powerful and their syntax may be both unfamiliar to and overwhelming for variability engineers. In contrast, a dedicated language for variability management, such as a delta language, may offer operations specifically tailored to expressing variability in the source language, e.g., to preserve consistency by avoiding dangling references.

7 Conclusion

In this paper, we presented a framework to create delta languages for source languages given as EMOF-based meta models to express variability in SPLs and SECOs. We illustrated how to derive syntax and semantics for custom delta languages from a source language's meta model. For this purpose, we defined semantics for six types of standard delta operations and illustrated how to analyze an EMOF-based meta model of a source language to find suitable instances of these operations. We used this information to define a delta dialect to a common base delta language in order to create a custom delta language. The generated delta languages are interoperable and integrate seamlessly into a common variant derivation mechanism to create products of an SPL or SECO for multiple source languages.

The case study showed that DeltaEcore can be applied to languages with different characteristics and that the automatically generated standard delta operations cover a wide range of suitable delta operations. However, it also suggested that a large number of excess

[3] `http://omg.org/spec/QVT/1.0`
[4] `http://eclipse.org/atl`
[5] `http://eclipse.org/epsilon/doc/etl`

delta operations is derived especially with opposite references. In our future work, we will consider how to reduce this number and how to identify delta operations particularly useful to variability engineers. We will further inspect how to integrate support for family-based analyses into DeltaEcore to allow efficient processing and comparison of analyses on multiple variants of an SPL or SECO. Finally, we plan to perform an industrial-scale case study with partners from the automotive sector using our tool DeltaEcore.

Acknowledgments

This work was partially funded by the European Social Fund (ESF) and the Federal State of Saxony within project VICCI #100098171.

References

[AKL13] Sven Apel, Christian Kästner, and Christian Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013.

[Bos09] Jan Bosch. From Software Product Lines to Software Ecosystems. In *Proceedings of the 13th International Software Product Line Conference*, SPLC, 2009.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[DL04] Josh Dehlinger and Robyn Lutz. Software Fault Tree Analysis for Product Lines. In *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*. IEEE, 2004.

[DS11] Ferruccio Damiani and Ina Schaefer. Dynamic Delta-Oriented Programming. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, page 34. ACM, 2011.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Proceedings of the 17th International Software Product Line Conference (SPLC)*, SPLC'13, 2013.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, page 4. ACM, 2013.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*, page 6. ACM, 2011.

[KLM03] Bernhard Kaiser, Peter Liggesmeyer, and Oliver Mäckel. A New Component Concept for Fault Trees. In *Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software-Volume 33*. Australian Computer Society, Inc., 2003.

[KW04] Tim Kelly and Rob Weaver. The Goal Structuring Notation–A Safety Argument Notation. In *Proceedings of the Dependable Systems and Networks Workshop on Assurance Cases*, 2004.

[Lev95] Nancy G Leveson. *Safeware: System Safety and Computers*. Addison-Wesley Longman, 1995.

[LSKL12] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In *Tests and Proofs*, pages 67–82. Springer, 2012.

[MVG06] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering - Foundations, Principles and Techniques*. Springer Berlin/Heidelberg, 2005.

[RW11] Bernhard Rumpe and Ingo Weisemöller. A Domain Specific Transformation Language. In *Proceedings of the Workshop on Models and Evolution (ME)*, 2011.

[SA13] Christoph Seidl and Uwe Aßmann. Towards Modeling and Analyzing Variability in Evolving Software Ecosystems. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, VaMoS'13, 2013.

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond*, pages 77–91. Springer, 2010.

[Sch10] Ina Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *VaMoS*, pages 85–92, 2010.

[SLFG09] Pablo Sánchez, Neil Loughran, Lidia Fuentes, and Alessandro Garcia. Engineering Languages for Specifying Product-Derivation Processes in Software Product Lines. In *Software Language Engineering*, pages 188–207. Springer, 2009.

[SRC+12] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software Diversity: State of the Art and Perspectives. *STTT*, 14, 2012.

[SSA13] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. Variability-Aware Safety Analysis using Delta Component Fault Diagrams. In *Proceedings of the 4th International Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*, FMSPLE'13, 2013.

[ZSS+10] Steffen Zschaler, Pablo Sánchez, João Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, and Uirá Kulesza. VML*–A Family of Languages for Variability Management in Software Product Lines. In *Software Language Engineering*, pages 82–102. Springer, 2010.