Structurally Evolving Component-Port-Connector Architectures of Centrally Controlled Systems

Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, Andreas Wortmann
Software Engineering, RWTH Aachen University
Aachen, Germany www.se-rwth.de

ABSTRACT

The increasing complexity of software variants demands for variation management techniques suitable for industrial practice. As "clone-and-own" with subsequent manual evolution still is a popular method to create software variants, this leads to product lines that are difficult to maintain and evolve and can produce conflicts when changes occur in both, product line and variant. Where general approaches to differencing and merging handcrafted changes to products perform suboptimally, respecting assumptions on the structure of the architecture can reduce the differencing search space and yield better results. We present a novel method for differencing and merging hierarchical component-port-connector architectures based on the FOCUS calculus. It leverages assumptions on the distribution of components to facilitate calculating differences between architecture versions and deriving delta bundles to update software products to changes in the underlying product line. Through a (preliminary) survey with 27 participants, we compared the results of our method with the results of manually differencing. The survey showed that this method yields deltas and merge results that are considered correct by the participants. Overall, we found that including assumptions about the architectural style and that grouping related deltas into compact bundles can greatly facilitate merging manually created and evolved products back into their underlying product lines.

1 INTRODUCTION

Software is devouring the world: most of the systems we use on a daily basis, the systems society relies upon, and industrial innovation are driven by software. This growing complexity of software drives industrializing software development. In particular, model-driven engineering (MDE) [15] aims to reduce the gap between the problem domain of discourse and the solution domain of software engineering. To this effect, it leverages more abstract and possibly domain-specific models as primary development artifacts. To support the modeling of software architectures, research and industry have developed over 120 [23] architecture description languages [25]. These languages have been successfully deployed to and used in various domains, such as automotive [18], avionics [13], or robotics [30]. However, in many of these domains, reuse of software (documentation, models, source code, *etc.*) among different products is a major challenge. Research investigates methods and concepts to harness variability modeling techniques [16], such as feature diagrams [5] or deltas [34], to arrange the different constituents of software products in Software Product Lines (SPLs) in such a way that their systematic reuse across different products is supported.

While these contributions enable systematic reuse of software development artifacts, the reality of software reuse often still relies on manually copying and adjusting artifacts (so-called "clone-and-own" [14]) methods. This severely complicates the evolution of products or product lines, as changes to individually evolved products may need to be re-integrated into the originating product line. Performing the necessary differencing and merging activities manually is costly and error-prone. Hence, there has been active research [24] in automating both activities for various kinds of software development artifacts, such as architecture views [1], component and connector architecture models [9], UML-based architectures [37], Simulink models [20], and many more.

Many general approaches to differencing and merging perform suboptimally due to solving overly generic challenges, whereas including assumptions about the used architectural style can reduce the differencing search space and, hence, produce useful results more efficiently. We present a novel method for the efficient differencing and merging of hierarchical component-port-connector (CPC) architectures [25] that are based on the FOCUS calculus [6]. This method rests on the assumption that the node degrees of CPC architecture graphs often follow a power-law distribution [4]. Based on this, it calculates related components and their differences between two versions of a CPC architecture and produces bundles of related deltas (*i.e.,* small architecture model transformations) that can be applied to update manual changes from a predecessor product architecture to a successor semi-automatically. Bundling related deltas relieves architecture modelers from comprehending large sets of minuscule deltas and it reduces errors as bundled deltas are applied en bloc, hence omitting necessary deltas is impossible.

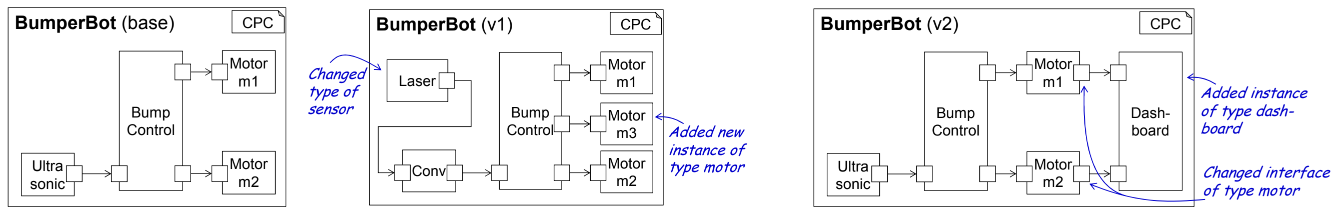
In the following, Section 2 introduces preliminaries and Section 3 presents a motivating example. Section 4 presents our method for computing differences in CPC architectures and Section 5 explains how the resulting deltas are used to merge architectures. Then, Section 6 presents a user study on the reception of our method's results regarding their use. Section 7 discusses our results. Section 8 highlights related work. Section 9 concludes.

2 PRELIMINARIES

We leverage delta modeling [10] to describe differences between architecture models that are based on the FOCUS calculus [6, 31]

This research has partly received funding from the German Federal Ministry for Education and Research under grant no. 01IS16043P. The responsibility for the content of this publication is with the authors.

VaMoS '20, February 5–7, 2020, Magdeburg, Germany
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7501-6/20/02…$15.00
https://doi.org/10.1145/3377024.3377035



(a) Base variant of the bumperbot architecture.

(b) First version (v1) evolved from the base architecture.

(c) Another version (v2) evolved from the base architecture.

Figure 1: Three versions of an architecture. Two versions evolved out of the same base.

for discrete software systems. Therefore, this section briefly recapitulates FOCUS, the definition of deltas following existing definitions [10, 21], and contains assumptions about the used CPC architectures.

2.1 FOCUS

FOCUS is a mathematical framework for describing software-intensive systems as networks of stream-processing functions (SPFs). To this end, the behavior of each component is defined by a function from its input channels (*i.e.,* a set of streams) to its output channels (also a set of streams). Channels transport messages of their respective types in order of their transmission only—they do not have behavior on their own. The stream-processing functions of FOCUS are mathematical objects that can be composed and refined [6]. Through this, *stepwise refinement*, the process of starting with a set of underspecified functions (requirements) and decomposing and refining these over time into fully specified functions (implementations) becomes possible and verifiable. For communication, FOCUS provides three different timing paradigms (time-synchronous, untimed, and timed). Out of these, we consider time-synchronous communication, which is typical for embedded systems, only. In time-synchronous systems, a global clock determines the progress of time and in each time slice, a single message is passed on every channel.

2.2 MontiArc

While FOCUS is a pure mathematical framework, we use its realization in the MontiArc ADL [8, 17, 33] to evaluate our method. In MontiArc, component types are either atomic or composed and contain subcomponents. The behavior of atomic components corresponds to a single SPF, whereas the behavior of composed components corresponds to the composition of functions. The subcomponents of composed components communicate via the interfaces of directed and typed ports defined in their respective component types. Following FOCUS, only components have behavior themselves, while connectors between components transmit messages according to the underlying timing paradigm only. More complex communication, *e.g.,* communication busses, would be modeled using components. MontiArc can be easily tailored to different domains [7] and we have used it in automotive [12], robotics [2], and education [32].

Figure 1 illustrates three MontiArc architectures that represent different versions of the decomposed `BumperBot` component type. Each of these yields subcomponents providing the functionality of sensors (such as `Ultrasonic`), of actuators (*e.g.,* `Motor`), or of system control (*e.g.,* `BumpControl`). Whether these are composed themselves is not visible from the outside and, hence, fosters reuse of components in different contexts.

2.3 Deltas

Instead of using deltas to derive products of a delta-oriented SPL, we use deltas to express the change operations needed to transform one model version into another. Explicitly, a change operation *op* is an atomic addition or removal of a model element, and a delta $\delta = \{op_1, \ldots, op_n\}$ is a set of change operations [21] that can be applied to a model m_i to transform it into a model m_j. The set of all possible deltas is denoted as $\mathcal{D}$.

Following [10], we define $-(-) : \mathcal{D} \times \mathcal{M} \to \mathcal{M}$, where $\mathcal{M}$ is the universal set of all models, to be the application function that applies a delta to a model. Given $\delta \in \mathcal{D}$ and $m \in \mathcal{M}$, $\delta(m) \in \mathcal{M}$ is the model resulting from applying delta δ to model m. Deltas can be chained by using the composition function $\cdot : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ [10]. Hence, $(\delta_y \cdot \delta_x)(m) = \delta_y(\delta_x(m))$ denotes first applying δ_x to model m and then applying δ_y. We derive deltas $\delta_1, \ldots, \delta_n$, such that $\delta_{m_1,m_2} = \delta_1 \cdot \ldots \cdot \delta_n$ is a regression delta [22] explicitly capturing the differences between arbitrary CPC architecture models. The deltas chained by a regression delta do not necessarily lead to valid models if applied individually.

3 EXAMPLE

A major challenge in the evolution of software is managing changes across different artifacts or versions of the same artifact. Changes in an artifact must be accompanied by corresponding changes to dependent artifacts to ensure consistency. When several developers are tasked with evolving software, their changes must be brought together and conflicting changes need to be resolved. We compare and merge versions of an architecture that evolved out of the same base version.

Consider the example in Figure 1 that shows three versions of the BumperBot CPC architecture. The example consists of the base version, as well as two versions (v1, v2) that were developed independently by two different teams and evolved from the base version. The base version consists of four components. The controller component of type `BumpControl` is connected via outgoing ports to two components of type `Motor` that control locomotion. The environment of the BumperBot is captured by an ultrasonic sensor and the corresponding component of type `Ultrasonic` that is connected to the controller via an incoming port. In the first version, the developers replaced the `Ultrasonic` component by a component of type `Laser`. To ensure consistency to the existing port of the controller component, the developers also added a component of type `Conv` that translates the signal of the laser component to a signal compatible with the controller's interface. The developers also added a third motor component to enhance mobility. In v2, the developers instead added a second port to the `Motor` type and connected the two motor components to a component of type `Dashboard` to provide a view onto the performance of the motor components.

When merging the two evolved versions v1 and v2, a decision must be made whether to keep the ultrasonic component or replace it with the laser component. The latter implies that the converter component also needs to be selected, as the interface of the laser component is incompatible with the controller component. Vice versa, taking the converter component without the laser component results in an ill-formed architecture, given that all ports need to be connected. In addition, extending the type `Motor` with the outgoing port from v2 also implies including the dashboard component in the merge result. However, the change to the `Motor` type in v2 is in conflict with the addition of a motor component in v1, as there is no viable option to connect the component to. A method for differencing and merging should entail both, identifying dependent changes and successively abstracting from minuscule changes, as well as identifying conflicting changes.

4 STRUCTURAL DIFFERENCING

In this section, we present an algorithm for finding deltas between two CPC architectures, *e.g.,* the base version and v1 from the example in Section 3. Combined, the identified deltas form a regression delta between the architectures. This algorithm is based on a set of assumptions that allow the algorithm to group the found deltas into more compact delta bundles.

The node degrees of many networks, both man-made and naturally created, follow a power-law distribution [4]. In other words, these networks contain a large number of nodes with only a small number of connections and a small number of nodes with a large number of connections. If we consider components to be nodes and connectors to be edges, this pattern can also be found in software architectures if one or multiple central controller components coordinate the data exchange between the other components. Many architectural patterns incorporate some form of such a central component, *e.g.,* bus-based architectures (which would model the bus as a component in MontiArc). Note that the central controller only needs to be at a central position in the architecture and ideally be well-connected to the other components. Despite its name, this component does not actually have to *control* other components.

Assumption A1. The architectures to be analyzed follow a structure in which one component acts as a central controller

This assumption allows us to derive contextual relationships between the components connected to the central controller. These contextual relationships are leveraged to create more complex replacement operations that can replace components by components with different names, types, or interfaces. We build those contextual relationships based on the port of the central controller that a group of components is connected to.

Assumption A2. If different groups of components are connected to the same port of the controller in different versions of the architecture, these groups offer the controller a related functionality.

This is a reasonable assumption as ports with different functionality often come with different names or types. In case the architects refactor parts of the architecture and connect different groups of components to the central component, they will likely also change the name or type of the port of the central component that the components are connected to. This assumption reduces the size of the search space in which we have to compare components for similarity. As we work on the ports of the central controller, this central controller needs to be present in both versions of the architecture. Using two different components as starting points for our differencing could lead to calculating incorrect deltas.

Assumption A3. The controllers found in the two compared versions of the architecture are different versions of the same component instead of two completely different controllers.

As already mentioned in [1], differencing two completely different architectures could be done by just deleting all nodes from the first and adding all nodes of the second architecture. While these deltas would be correct, we would not consider them useful. Therefore, it is reasonable to assume a certain level of similarity between the compared architectures. The assumption of comparing two similar architectures is also made by related architecture differencing algorithms such as [1].

Assumption A4. We compare two versions of the same architecture instead of two completely different architectures.

The assumptions **A1** - **A4** allow our differencing algorithm to produce deltas that do not only add or remove components but also replace groups of components. Our algorithm does not produce incorrect deltas when comparing architectures that do not follow our assumptions about the architectural style (**A1**- **A2**). However, the derived contextual relationships are less useful in this case, *i.e.,* large groups of components would be replaced by each other even though they might not be contextually related.

4.1 Algorithm Description

In general, the differencing algorithm consists of six steps:

(1) Identify the controller components in both versions of the architecture.
(2) Map the ports of the controller components to each other.
(3) Find the tree structure of components that is connected to each port. If the graph of connected components contains a circle, the spanning tree starting at the controller component is chosen.
(4) Map the components of the trees identified in the previous step to each other.
(5) Create deltas from component mappings.
(6) Bundle related deltas that should not be applied individually.

These steps are repeated for each hierarchy level, *i.e.,* each subcomponent of a composed component, if two subcomponents in different versions of the architecture are recognized as unequal versions of each other. This leads to creating high-level deltas, which do not require the user to understand low-level change operations. For example, a delta that deletes a composed component only captures which component it deletes but does not include the individual change operations to delete every element within the composed component.

Step 1: Identify the controller components. First, we reduce the search space by limiting the search for controller components to the common subgraph induced by the components that are part of both architectures. Even if the real controller component was only added in a later revision of the architecture as the architecture got more complex, it makes more sense to use two versions of the same component as a starting point for further analysis. Using different components as controller components could lead to calculating incorrect deltas.

Next, the component that has the largest number of connections to other components is searched within the common subgraph. Since the common subgraph only contains components that exist in both versions of the architecture, connections to or from added or removed components are not considered in the calculation. In the case that the common subgraph does not contain any connections, all connectors of the first version of the architecture are considered in the calculation. In the case that multiple components have the same number of connections in the common subgraph, the algorithm sums up the distances between these components and every other component of the common subgraph. The algorithm chooses the component that has the smallest sum of distances as controller. This is done to ensure the central component has a central position in the architecture.

Step 2: Map the ports of the controller components to each other. The mapping of ports relies on multiple confidence levels. First, the algorithm tries to match ports by searching for ports with matching names and directions. If there are ports left for which no counterpart could be found, the algorithm next searches for ports with matching data type and direction. All ports that still cannot be matched to a counterpart are considered new.

Step 3: Find spanning trees starting at the ports of the controller component. As mentioned above, we assume that groups of components that are connected to the same port of the controller in different versions of the architecture offer a related functionality. Therefore, the algorithm builds spanning trees starting at each port of the controller to find these groups of components. The main idea of the spanning tree search is to start Breadth-First Searches (BFSs) at the ports of the controller component. For each component connected to a port of the controller component, the algorithm starts a BFS at the component that is directly connected to the controller component. Each BFS only considers connections pointing in the same direction, *i.e.,* either away from or towards the already found components. The direction is determined by the direction of the connection between the controller component and its direct neighbors. As an exception to this rule, connections are also included in the BFS if the components they add to the spanning tree would otherwise not be found. Components can be part of multiple spanning trees. This is resolved in step five by bundling deltas that should not be applied individually.

Step 4: Map the components of the trees identified in the previous step to each other. Spanning trees that are connected to matching ports of the controller components are considered to belong together. In this step, the algorithm matches the contents of spanning trees that belong together.

Within each pair of trees, the algorithm first searches for components that have the same name and neighbors with the same names. Similar to UMLDiff [37], these components serve as “landmarks” to later recognize other components by their structural relations to these landmarks. However, unlike UMLDiff, our approach also considers the names of the neighbors of each landmark. If the names of the neighbors do not match, a component cannot be considered to be a landmark. This prevents choosing moved components as landmarks. Each landmark represents a group of currently one component, which can be extended in the following to create bigger landmarks.

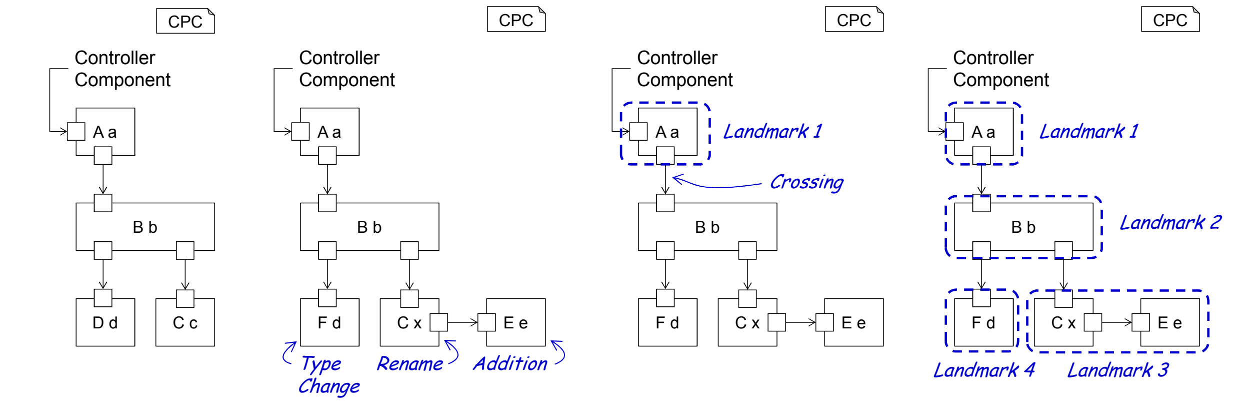
Next, all components that are not selected as landmarks are matched by their relationship to the already existing landmarks. For this purpose, the algorithm first creates so-called “crossings”. Each crossing connects a landmark to a component that has not yet been matched to its counterpart. Afterward, the algorithm tries to match these crossings to each other. This is done by considering the following three cases:

(1) There is exactly one crossing found in each architecture. Therefore, the components that are not already matched can be matched and form a new landmark.
(2) There is one crossing found in the one architecture and no crossing in the other architecture. The component that is not already matched gets added to the already existing landmark it is connected to.
(3) There are multiple crossings found in both architectures. Here, the algorithm tries to match by the type of components. If the crossings do not connect components of the same types, it tries to match components for which at least one component type matches.

Figure 2 illustrates this procedure using the architectures in Figure 2(a) and Figure 2(b). Initially, only component a is selected as a landmark, because the neighbors of components b, c, and d do not match. Next, b is chosen as a second landmark according to case 1. Then, component x from Figure 2(b) is mapped to component c from Figure 2(a) because they share the same type. After the crossing $b \rightarrow x$ is resolved, there is only one crossing in each version of the architecture left. Therefore, case 1 is applied and component d forms a new landmark. Lastly, component e is merged into the landmark of component x because there is no corresponding crossing found in the first architecture. Accordingly, two replacements are created: $[C\ c] \rightarrow [C\ x, E\ e]$ and $[D\ d] \rightarrow [F\ d]$.

Step 5: Create deltas from component mappings. Now that we have groups of matching components, the algorithm can create deltas from these mappings. First, the algorithm creates *replacements* from all component groups that could be matched in the previous steps. Then the algorithm creates *deletions* for all components from the first version of the architecture that could not be matched and *additions* for all components from the second version of the architecture that could not be matched.

Step 6: Bundle related deltas that should not be applied individually. Some of the deltas created in the previous step may lead to malformed configurations. For example, if the architectures from



(a) Excerpt of the first version of an architecture.
(b) Excerpt of the second version of an architecture.
(c) Initial landmarks by matching name and neighbors in version two of the architecture.
(d) Final landmarks in version two of the architecture.

Figure 2: Example for the structural mapping performed during step 4.

Figure 1(a) and Figure 1(c) are compared to each other, two replacements are created. Each of them replaces one of the motors by the motor and the dashboard. However, applying only one of the deltas would result in an architecture that contains a dashboard with only one port. Therefore, the algorithm bundles those two replacements so that it is not possible to create an invalid dashboard, *i.e.,* one with only one port, by applying only a subset of the created deltas. Note that this step is not technically necessary. Instead, it provides an error-prevention mechanism for human developers that decide which deltas to apply when merging architectures. We choose which components to bundle by looking at the types of components that are affected by them. Deltas that affect one or more components of the same type are bundled.

Formally, a delta bundle is a composite delta (*cf.* [10]) $\delta_m = \delta_1 \cdot \ldots \cdot \delta_k$ with respect to a model m, such that applying the delta bundle to m yields a valid model, *i.e.,* for a delta bundle δ_m it holds that $m \in \mathcal{M}_L \implies \delta_m(m) \in \mathcal{M}_L$, where $\mathcal{M}_L \subseteq \mathcal{M}$ is the set of all models valid to the specification of modeling language L.

5 DELTA-BASED ARCHITECTURE MERGING

SPLs enable developers to create a common product line architecture that should be used among multiple variants of a product. Developers derive architectures for concrete products by configuring which features they want to include in the product. As all generated artifacts, the derived concrete architecture ideally should not be modified manually. However, in practice developers may make modifications to the derived architectures [11, 35]. This may lead to situations in which the product line architecture and the derived architecture are modified simultaneously. Updating a variant to changes in the product line while preserving local changes requires sophisticated techniques. In [35], the authors propose a method for updating variants using a “Three-Way-Merge”.

Three-Way-Merges compare three different versions of an artifact. In this case, the unmodified base version (b) of the product line architecture is compared to the modified product line architecture (v_1) and the modified derived architecture (v_2). In general, one base version is compared to two different modifications of the base version. Merging the changes from both modifications cannot be done fully automated in many cases [35]. Therefore, we provide a way of merging three architectures, *i.e.,* one base version and two modifications to the base version, with the assistance of a developer.

However, merging architectures is an error-prone process. As mentioned above, not all deltas may be applied individually because doing so could lead to a malformed architecture. Moreover, deltas may be contradictory and, therefore, not all combinations of deltas may be applied. Therefore, we assist the developer by bundling deltas that should not be applied individually (*cf.* Step 6 in Section 4) and by detecting merge conflicts between deltas. This process is shown in Figure 3. We consider two deltas that replace or delete the same component of the original architecture to be in conflict with each other. If the developer decides to merge two deltas that are in conflict, we inform the developer about the conflict and ask the developer to choose between the conflicting deltas. To reduce the extraneous cognitive load (*cf.* [28]), we provide deltas on a high level of abstraction, *i.e.,* the developer is only told which component groups should be added, removed, or replaced. Ports and connections are hidden from the developer.

After the user has selected a set of deltas that can be used to create a well-formed architecture, the deltas are applied to the base architecture. Formally this corresponds to applying $(\delta_1 \cdot \ldots \cdot \delta_k)(b)$, where $\delta_1, \ldots, \delta_k$ are the deltas selected by the user. The deltas $\delta_1, \ldots, \delta_k$ are also part of the regression delta δ_{b,v_1} or δ_{b,v_2}. Next, the result is exported to the original format. The deltas are applied in hierarchical order, *i.e.,* deltas modifying a composed component are applied before deltas that modify its subcomponents.

Abstractly, the inner structure of each component is modeled as a graph structure. Each instance of a subcomponent represents a node in the graph and each connector represents an edge. This structure allows us to apply deltas by modifying the graph. Additions and deletion deltas can be realized by adding or removing nodes and their attached edges. Note that if a component is between two other components, *e.g.,* component B in $A \rightarrow B \rightarrow C$, the disappearance

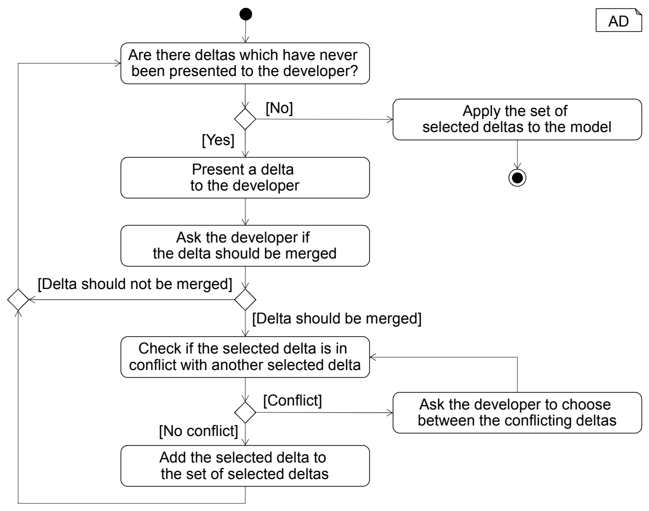

AD
Are there deltas which have never been presented to the developer?
[No]
Apply the set of selected deltas to the model
[Yes]
Present a delta to the developer
Ask the developer if the delta should be merged
[Delta should not be merged]
[Delta should be merged]
Check if the selected delta is in conflict with another selected delta
[Conflict]
Ask the developer to choose between the conflicting deltas
[No conflict]
Add the selected delta to the set of selected deltas

Figure 3: Process of requesting a set of deltas to apply from the user.

of this component would not be modeled as a Deletion but instead as a Replacement to avoid having unused ports.

Replacements are realized by first removing a (sub)tree structure from the graph. Then, the tree that should replace the former tree structure is added. Lastly, we need to connect the root component of the tree to the rest of the graph. If only one component is replaced, we further consider the ports of that component. We use the same algorithm described in step two of Section 4 to decide which ports to replace with which other ports. The component that replaces the previous component may not have all ports that were used by the previous component. To avoid creating dangling connectors that connect components to non-existent ports, we add all unmatched ports of the replaced component to the new component.

After all selected deltas are applied, we export the architecture by using the pretty-printer provided by MontiArc but instead of printing the actual subcomponents, inner components and connections of components, the contents of the graph structure are printed. This allows us to export the architecture in the same format that it was provided in, while also allowing us to replace the inner structure of each component.

In summary, the overall merging process consists of four steps:

(1) Compare the base architecture to two modifications of the base architecture and create deltas (*cf.* Section 4).
(2) Assist the developer in selecting a valid set of deltas, *i.e.,* a set of deltas that will produce a well-formed result (*cf.* Figure 3).
(3) Apply the set of deltas selected by the developer to the internal representation of the base architecture.
(4) Export the architecture.

6 USER STUDY

We conducted a preliminary user study to investigate whether our method for differencing and merging of CPC architectures yields results (1) that are considered correct; (2) that compare to manually merging software architectures; and (3) that provide intuitive benefits. This manifests in the following three research questions:

- **RQ1:** Do the kinds of deltas (add, delete, replace, *etc.*) intuitively created by entry-level software practitioners match the kinds of deltas used in our approach?
- **RQ2:** Are the identified deltas expressing differences between two architecture variants considered correct?
- **RQ3:** Is the merging result of architecture variants considered correct?

We focus our research questions on entry-level software practitioners for multiple reasons: (1) They are the least likely to be preoccupied with (types of) deltas produced by other tools and, thus, can give appropriate "intuitive" deltas (**RQ1**) (2) Their lack of familiarity with differencing and merging makes them a likely group of software practitioners who likely do not understand our results and, therefore, consider them incorrect (**RQ2**-**RQ3**) (3) They are easier to recruit for a user study than highly experienced software practitioners. Since our research questions aim at evaluating opinions, we chose a survey as the evaluation type and handed out evaluation tasks and questioners to willing participants to collect data.

6.1 Survey

The survey was executed through a handed out exercise sheet that was designed for answering our research questions and consisted of manual tasks for differencing and merging of architecture variants, as well as a mix of open-ended and closed questions. The exercise sheet consisted of four tasks:

(1) The participants were asked to reconstruct the differences between architecture variants developed by two different teams and to express changes between these variants and a common base as deltas. The architecture variants and the common base were provided both as graphical and textual models. The architectures were the same as those shown in Figure 1 with the exception that no third motor was added in the first version.
(2) The participants were asked to use the deltas they identified in the first part to merge the two variants and provide the resulting architecture as a graphical model, as well as the list of deltas used for merging.
(3) The participants were provided with the differencing and merging result of our method. This result included one delta bundle consisting of two deltas and a short explanation of what a delta bundle is. Participants were asked to compare the result of our method to their own results and to explain their assessment. Part three offered a free text area to answer the questions.
(4) The participants were asked closed questions through a questionnaire.

We provided all participants with printed exercise sheets and asked them to complete the tasks and answer the questions by a specific date. Of those who received an exercise sheet, 27 completed all tasks and returned the completed exercise sheet.

6.2 Threats to Validity

Our study is subject to threats to construct validity and external validity. To limit the effects of the threats evaluation apprehension and hypothesis guessing on construct validity, we guaranteed the participants complete anonymity, informed them about the goal and procedure of the study, and asked for honest answers. However, these threats cannot be completely dismissed. Also, it is not possible to exclude that participants misunderstood questions. We, therefore, face threats to conclusion validity. Furthermore, our study faces threats to external validity as the participants in the study mostly have, giving our working context, a university background. The study population can be called a convenience sampling, as participants were mostly recruited from students and fellow software engineering researchers, who were encouraged to also recruit their students as well. Through threats of selection bias and convenience sampling, the final study population may not be representative of the global population of software developers. As the study did not include any experienced developers, our results must be considered preliminary. Moreover, the participants' self-assessment on their experience is prone to the Dunning-Kruger effect.

6.3 Observations

A total of 27 participants completed all tasks and questions in the study. The participants were asked three closed questions about their background experience. Answers to closed questions could be given through a 5-point Likert scale, with answers ranging from 1 (no experience or strongly disagree) to 5 (high experience or strongly agree). Namely participants were asked about their experience with software architectures ($x_{MOD} = 3$, $\overline{x} = 3.22$, $s = 0.97$), experience with model differencing ($x_{MOD} = 2$, $\overline{x} = 1.89$, $s = 0.70$), and experience with merging ($x_{MOD} = 3$, $\overline{x} = 2.93$, $s = 1.00$). 11 of the participants (41%) stated that they have substantial experience (4 or 5) with software architectures. The respondents answered that they have low experience with model differencing, as nobody stated that they have above medium (3) experience.

To answer research question **RQ1**, the participants manually performed their own differencing of the provided architecture variants and compared their results to our solution. Of the respondents, 18 provided at least basic *add* and *delete* operations, 7 only provided *replacement* operations, and two provided high-level descriptions. Of those who provided basic add and delete operations, two also provided replacement operations and three provided additional operations such as *modify*, *define component type*, and *create inner component*.

Regarding research question **RQ2**, the participants were tasked to compare their differencing solutions to the solution we provided and asked whether they consider the deltas identified in our solution to be correct. The deltas provided by our method included one delta bundle consisting of two deltas. In a closed question 20 (74%) of the respondents agreed or fully agreed that the deltas identified by our method are correct ($x_{MOD} = 5$, $\overline{x} = 4.12$, $s = 1.03$). The participants also agreed with our differencing solutions in the open text questions, but two respondents stated that the compactness of our solution made some changes more difficult to identify and that they would prefer more detailed solutions.

Research question **RQ3** focused on merging architecture variants. To answer the research question, the participants were asked to compare their merging solutions with the solution we provided, and, in closed questions, if they consider our solution to be correct.Of the respondents, 25 (82%) agreed or fully agreed that the provided solution is correct ($x_{MOD} = 5$, $\overline{x} = 4.77$, $s = 0.51$).

7 DISCUSSION

Our approach to differencing and merging hierarchical CPC architectures incorporates assumptions about the architectural style (**A1**-**A2**) to facilitate matching architectural elements and hence produces meaningful results more efficiently. These assumptions restrict our algorithm to comparing multiple versions of the same architecture as it occurs when developing an SPL. If semantically different groups of components are connected to the same ports of the central controller in different versions of the architecture (**A2** not met), the algorithm will only produce trivial or counterintuitive deltas, *e.g.,* very small add/delete operations or very large replacement operations that replace unrelated functionalities. If our algorithm operates on two completely different input architectures not coming from the same product line (**A3** and **A4** are not met) it will either produce trivial or incorrect results. If two versions of the same component are identified as controller component but the architecture overall is very different (**A4** not met), the algorithm will produce trivial results. If two completely different components are assumed to be different versions of the central controller (**A1/A3** not met), the algorithm will produce incorrect deltas, *i.e.,* deltas where $\delta_{m_1,m_2}(m_1) \neq m_2$. We consider this assumption of matching at least one pair of different versions of the same controller component to serve as a starting point for our further differencing to be reasonable as we still allow the matched controller components to have different interfaces and behaviors.

As with other approaches, our goal is to produce results comparable to those of manual differencing and merging processes. Different participants identified different kinds of deltas (**RQ1**). According to their solutions, it should be considered whether providing not only generic add and delete operations but also more complex change operations could be useful. By leveraging delta modeling we are able to express differences between two architecture versions as a set of change operations needed to transform one architecture into the other. Through grouping of related change operations we then abstract from minuscule changes. Furthermore, we bundle deltas that lead to invalid intermediate architecture models if applied one-by-one. This grouping and bundling can simplify their use and reduce errors, as deltas and their dependent deltas are applied en bloc during the merging process. A majority of the study's participants showed acceptance for the deltas identified by our approach (**RQ2**). However, as there is a greater deviation between the responses, the abstraction from more fine-granular change operations has not been accepted equally by all participants (**RQ1/2**). This is supported by participants explicitly having asked for more fine-granular deltas. Most of the participants agree with our solution (**RQ2**). This implies that grouping change operations and bundling deltas can be useful as it reduces the number of change operations that have to be considered while still providing correct results. In some cases, however, control over more fine-grained change operations may be useful. To this end, we support control over concrete change operations only on request.

Most of the participants also showed acceptance for our merging solutions (**RQ3**). This insight is reinforced by the fact that a large portion of the solutions of the manual merging is similar to the solution of our approach. We conclude that our approach indeed produces results that are considered correct and comparable to manual solutions (**RQ2/3**). However, the opinion of the study participants does not necessarily represent the opinion of the global population of software practitioners, as participants were recruited from students and faculty members.

8 RELATED WORK

Software merging has been excessively studied in the past [26]. However, while there is a wealth of techniques and tools for merging programs (*i.e.,* source code), support for model evolution has been identified as a more recent challenge [27]. Model differencing, not only as a prerequisite for model merging, has become an independent field of research and numerous model differencing approaches have emerged [19], including identity matching [3], signature-based matching [29], similarity-based matching [36], and semantic differencing [24]. Besides more general approaches, there exist matching algorithms tailored to particular modeling languages that incorporate knowledge about the languages' semantics [27].

UMLDiff [37] is a structural-differencing algorithm for class diagrams that takes into account the UML semantics. Similar to our approach, input models are converted to directed graphs and nodes of the same conceptual entity are compared. The identification of conceptually same entities is based on a name similarity and the relationship to other entities. Concepts comparable to UMLDiff are implemented in DSMDiff [20], which also uses graphs as underlying data structure on which the differencing algorithm operates. This algorithm successively traverses the hierarchy of the input models while using signature and structural similarity matching to identify matches and differences. The approach, however, is applicable to domain-specific models in general and meta-model independent. A differencing approach specifically tailored to Component & Connector (C&C) architectures has been proposed in [1]. This approach is also based on tree matching, but can only find one-to-one mappings of components. [1] bases its delta finding algorithm on the cost of transforming the names of the nodes in the compared trees into their counterpart in the other tree. Like [1], we also use a name-based search to find starting points, *i.e.,* to create the initial *landmarks* (*cf.* Section 4), but our algorithm also takes the connections of components to their neighbors into account to find groups of related components. This allows our algorithm to find more complex replacement operations such as replacing one component by multiple components instead of the one-by-one replacements of [1]. However, [1] can also detect hierarchical move operations that are not considered by our algorithm. As future work, the two algorithms could be combined by using [1] to find the differences in the hierarchy of components while using our algorithm to compare components on the same hierarchy level.

Developed for message-driven C&C architectures, a method for semantic differencing [9] enables comparing the behaviors of components. The method transforms architectures into Büchi automata before proving refinement for component versions which yields counter-examples (witnesses) for non-refining component pairs.

In contrast to the presented approaches, our algorithm assumes an identifiable controller component from which we can span subtrees in order to reliably detect matches and content-related changes. This allows us to abstract from fine-grained change operations and thus increase the ease of use while avoiding errors by bundling interdependent deltas.

9 CONCLUSION

We have presented a novel method to calculate differences between different model versions of hierarchical component-port-connector architectures that assumes a power-law distribution to identify sets of related changes. These sets of related changes are represented as abstract deltas that support understanding and application of differences. Further bundling of deltas prevents the creation of invalid models. To investigate the perceived correctness of the differences, we conducted a survey with 27 participants. The results show that abstracted deltas are considered correct to compare architecture models and our three-way merge based on deltas produces results as expected by the participants. Overall, we found that leveraging the domain-specific consideration of the power-law distribution as a basis for calculating model differences as delta bundles can improve differencing and merging of architecture variants.

REFERENCES

[1] M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl, and D. Garlan. 2006. Differencing and Merging of Architectural Views. In *21st IEEE/ACM International Conference on Automated Software Engineering*. 47–58.

[2] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2016. Model-driven separation of concerns for service robotics. In *Proceedings of the International Workshop on Domain-Specific Modeling*. ACM, 22–27.

[3] Marcus Alanen and Ivan Porres. 2003. Difference and Union of Models. In *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*. Lecture Notes in Computer Science, Vol. 2863. Springer, Berlin and Heidelberg, 2–17.

[4] Réka Albert and Albert-László Barabási. 2002. Statistical Mechanics of Complex Networks. *Reviews of Modern Physics* 74 (Jan 2002), 47–97. Issue 1.

[5] Don Batory. 2005. Feature models, Grammars, and Propositional Formulas. In *International Conference on Software Product Lines*. Springer, 7–20.

[6] Manfred Broy and Ketil Stølen. 2001. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg.

[7] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2017. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *Modelling Foundations and Applications (ECMFA'17), Held as Part of STAF 2017*. Springer International Publishing, 53–70.

[8] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2017. Architectural Programming with MontiArcAutomaton. In *In 12th International Conference on Software Engineering Advances (ICSEA 2017)*. Athens, Greece, 213–218.

[9] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2017. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture*. IEEE, 145–154.

[10] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. 2010. Abstract Delta Modeling. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE '10)*. ACM, New York, NY, USA, 13–22.

[11] P. Clements and L. Northrop. 2002. *Software Product Lines: Practices and Patterns*. Addison-Wesley. 159 pages.

[12] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. 2019. SMArDT modeling for automotive software testing. *Software: Practice and Experience* 49, 2 (February 2019), 301–328.

[13] Peter H. Feiler and David P. Gluch. 2012. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley.

[14] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 391–400.

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

[16] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. 2001. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*. IEEE Computer Society, Washington, DC,

USA, 45.

[17] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. 2012. *MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems*. Technical Report AIB-2012-03. RWTH Aachen. http://aib.informatik.rwth-aachen.de/2012/2012-03.pdf

[18] Ramin Tavakoli Kolagari, DeJiu Chen, Agnes Lanusse, Renato Librino, Henrik Lönn, Nidhal Mahmud, Chokri Mraidha, Mark-Oliver Reiser, Sandra Torchiaro, Sara Tucci-Piergiovanni, Tobias Wägemann, and Nataliya Yakymets. 2015. Model-Based Analysis and Engineering of Automotive Architectures with EAST-ADL: Revisited. *International Journal of Conceptual Structures and Smart Applications* 3, 2 (July 2015), 25–70. https://doi.org/10.4018/IJCSSA.2015070103

[19] Dimitrios S. Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F. Paige. 2009. Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models (CVSM '09)*. IEEE Computer Society, Washington, DC, USA, 1–6.

[20] Yuehua Lin, Jeff Gray, and Frédéric Jouault. 2007. DSMDiff: A Differentiation Tool for Domain-Specific Models. *European Journal of Information Systems* 16, 4 (Aug 2007), 349–361.

[21] Sascha Lity, Mustafa Al-Hajjaji, Thomas Thüm, and Ina Schaefer. 2017. Optimizing Product Orders Using Graph Algorithms for Improving Incremental Product-line Analysis. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '17)*. ACM, New York, NY, USA, 60–67.

[22] Malte Lochau, Sascha Lity, Remo Lachmann, Ina Schaefer, and Ursula Goltz. 2014. Delta-oriented Model-based Integration Testing of Large-scale Systems. *Journal of Systems and Software* 91 (2014), 63–84.

[23] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. 2013. What Industry Needs from Architectural Languages: A Survey. *IEEE Trans. on Software Engineering* (2013).

[24] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2010. A Manifesto for Semantic Model Differencing. In *Proceedings International Workshop on Models and Evolution (LNCS 6627)*. Springer, 194–203.

[25] Nenad Medvidovic and Richard N Taylor. 2000. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* (2000).

[26] T. Mens. 2002. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering* 28, 5 (May 2002), 449–462.

[27] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. 2005. Challenges in Software Evolution. In *Eighth International Workshop on Principles of Software Evolution*. 13–22.

[28] Sharon Oviatt. 2006. Human-centered Design Meets Cognitive Load Theory: Designing Interfaces That Help People Think. In *Proceedings of the 14th ACM International Conference on Multimedia (MM '06)*. ACM, New York, NY, USA, 871–880.

[29] Raghu Reddy, Robert France, Sudipto Ghosh, Franck Fleurey, and Benoit Baudry. 2005. Model Composition - A Signature-Based Approach. In *Aspect Oriented Modeling (AOM) Workshop*.

[30] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. 2015. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics* 6, 1 (2015), 33–57.

[31] Jan Oliver Ringert and Bernhard Rumpe. 2011. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics* 5, 1-2 (July 2011), 29–53.

[32] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. 2017. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In *International Conference on Software Engineering: Software Engineering and Education Track (ICSE'17)*. IEEE, 127–136.

[33] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. 2013. A Case Study on Model-Based Development of Robotic Systems using MontiArc with Embedded Automata. In *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme*, Holger Giese, Michaela Huhn, Jan Philipps, and Bernhard Schätz (Eds.). 30–43.

[34] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond*.

[35] Sandro Schulze, Michael Schulze, Uwe Ryssel, and Christoph Seidl. 2016. Aligning Coevolving Artifacts Between Software Product Lines and Products. In *Proceedings of the 10th International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 9–16.

[36] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. 2007. Difference Computation of Large Models. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 295–304.

[37] Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: An Algorithm for Object-Oriented Design Differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 54–65.