Co-evolution of Simulink Models in a Model-Based Product Line

Robbert Jongeling
Mälardalen University
Västerås, Sweden
robbert.jongeling@mdh.se

Antonio Cicchetti
Mälardalen University
Västerås, Sweden
antonio.cicchetti@mdh.se

Federico Ciccozzi
Mälardalen University
Västerås, Sweden
federico.ciccozzi@mdh.se

Jan Carlson
Mälardalen University
Västerås, Sweden
jan.carlson@mdh.se

ABSTRACT

Co-evolution of metamodels and conforming models is a known challenge in model-driven engineering. A variation of co-evolution occurs in model-based software product line engineering, where it is needed to efficiently co-evolve various products together with the single common platform from which they are derived. In this paper, we aim to alleviate manual efforts during this co-evolution process in an industrial setting where Simulink models are partially reused across various products. We propose and implement an approach providing support for the co-evolution of reusable model fragments. A demonstration on a realistic example model shows that our approach yields a correct co-evolution result and is feasible in practice, although practical application challenges remain. Furthermore, we discuss insights from applying the approach within the studied industrial setting.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software evolution**; **Model-driven software engineering**.

KEYWORDS

Co-evolution, Change propagation, Clone management, Model-Driven Engineering, Software Product Line Engineering

ACM Reference Format:
Robbert Jongeling, Antonio Cicchetti, Federico Ciccozzi, and Jan Carlson. 2020. Co-evolution of Simulink Models in a Model-Based Product Line. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3365438.3410989

1 INTRODUCTION

When Darwin proposed his theory of evolution by natural selection, he famously concluded that "from (so) simple a beginnings endless forms most beautiful and most wonderful have been, and are being *evolved* [8]." In software engineering today, gradual and parallel changes are applied to software models with the goal of spawning variants addressing diverse requirements. The individual evolution and collective co-evolution of these variants need to be managed to ensure continued opportunities for reuse. To this end, software product line engineering (SPLE) proposes to organize development artifacts and their variants in product lines [20]. In this work, we study an industrial setting with a model-based product line where Simulink models are used to design and implement software components for the development of complex embedded systems.

Simulink is one of the most-used tools for model-based development of embedded systems [13]. It is a MATLAB-based graphical modeling environment with extensive support for simulations and code generation. Simulink models are created by defining a data flow by linking predefined and custom blocks. These blocks can represent defined functions such as logical operators or arithmetic operations, but also more complex functions such as integrators or look-up tables. A special type of block, called subsystem, can, in turn, contain a set of connected blocks, thereby providing the possibility of hierarchically organizing models.

In the studied industrial development setting, reuse of (parts of) models for software components is promoted to reduce the lead time for their development and maintenance. A known best practice for managing variants in a product line is through feature models, which shows the different variants present in the product line and the points at which the product can vary. Nevertheless, a commonly observed approach in industrial settings is to skip the creation of a feature model and instead start by copying assets from an existing project to reuse them in another project, leading to so-called *clone-and-own* product lines [24]. This kind of reuse is commonly used in industry because it requires no initial investment and it is simple to start with. Its downside is the lack of systematic reuse, which is required to fully benefit from the advantages of product lines [11].

In the SPLE paradigm, common functionality is contained in a *platform* from which various related products can subsequently be derived. Different products can thus be branched off from the platform and further developed to fulfill their unique requirements. This setup allows for customization of individual products while benefiting from organized reuse of common functionalities, which can centrally evolve. Indeed, a platform is expected to be periodically revised, for example, to fix bugs, or to include software for new or changed requirements. Such an evolution in the platform may need to be propagated to the derived products to keep them consistent with the platform. In other words, the derived products may need to co-evolve.

In typical model-driven engineering (MDE) scenarios, co-evolution refers to the need to update models upon a change to the metamodels they conform to. Automation of co-evolution of metamodels and models may utilize these conformance relationships [6]. This is one of three relationships typically considered in co-evolution. The other two are (1) a relation between a model transformation and a metamodel, and (2) an indirect dependence of a model on a metamodel [10]. In this paper we consider the need for derived

MODELS '20, October 18–23, 2020, Virtual Event, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410989

products to co-evolve, upon an evolution of the platform, to maintain the integrity of the product line. In the studied setting, derived products and platform are all Simulink models.

The setting differs from metamodel-model co-evolution due to the following three reasons:

(1) the relation between the models is not well-defined (at some point in the past, the derived product was branched off from the platform and after that has possibly undergone separate revisions, as illustrated in Figure 1);
(2) there is no traceability of reuse, so it is not known which portions of which artifacts should co-evolve; and
(3) derived products are not necessarily co-evolved, the changes in the platform might not be adopted depending on the requirements for that specific derived product.

In the studied industrial setting, analyses to find re-used model portions and assessment of the impact of propagating changes between them are currently performed manually. It is a time-intensive, difficult, and error-prone task. The choice on whether to propagate co-evolution changes to a product is based on the nature of the change, how the product differs from the platform, and on the specific requirements of the product. Some of the engineers working on the Simulink models are in charge of making this decision, we refer to them as domain experts. We aim to provide automated support for domain experts, at the granularity of reused model fragments, i.e. a group of connected model elements. This paper contributes an approach and insights from its application on real models. Furthermore, some of the proposed techniques may contribute to moving from clone-and-own approaches to product lines

The remainder of this paper studies the evolution of the platform and how to co-evolve the derived products. Section 2 describes the context of the work in terms of the studied industrial setting and describes the studied problem. Our proposed approach is outlined in Section 3, its implementation and a feasibility study in the form of its application on a realistic example are presented in Section 4. Insights from this process as well as the application of the approach in an industrial setting are discussed in Section 5. Related research works are listed in Section 6 and the paper is concluded in Section 7.

2 MOTIVATION

2.1 A clone-and-own Software Product Line

The studied setting is a development team at our industrial partner, responsible for the design, implementation, and testing of control software for a set of embedded systems. To enhance the opportunities for reuse of software components and their test cases, the team has adopted SPLE. Their product line is organized as one *platform*, which contains common functionality for products, and several derived products that are branched off from the platform throughout the revision history and then further developed, as illustrated in Figure 1. When a new requirement comes in, domain experts decide whether it is cross-product, and therefore to be implemented in the platform, or if it is specific to one of the products, and therefore to be implemented in that specific product only. New revisions of the platform are released periodically. Upon such a release, the new or changed functionality in the platform may need to be propagated to the derived products.

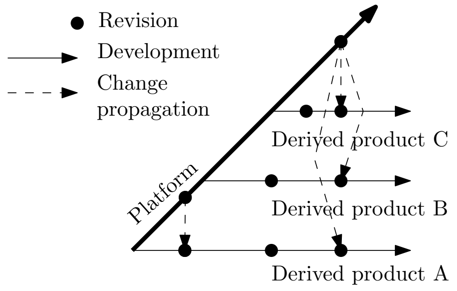


Figure 1: Organization of product line in a platform and derived products (A, B, C). Upon a new release of the platform, changes made in the platform since the last revision may need to be propagated derived products.

Currently, design and implementation of control software are done by the same team, using model-based development. The developed products are comprised of software components that are implemented in *Simulink* models. Each of these models is associated with a test harness and test cases. The eventually deployed C code is automatically generated from these models using *Embedded Coder*. Real-time behavior of the generated code is studied in hardware-in-the-loop tests, which are later followed by tests on lab hardware, and eventually on the real deployment target. In this use case, we specifically focus on the development of software components and their test cases.

It is worth noting that not all typical SPLE practices have been adopted in the studied setting, for example, there is no feature model, nor does the development make use of overloaded (150%) models. Overloaded models are models that contain a combination of all possible alternatives and from which a particular variant can be obtained by stripping away the irrelevant elements. Instead, the studied setting can be characterized as a clone-and-own product line, in which the platform contains re-usable components that are copied to the derived products. Typically, a component in a derived product is a reduced version of that component from the platform, or it is cloned, i.e., a one-to-one copy. However, in addition to the reduced and cloned components, derived products can also contain "new" components that do not exist in the platform and are only relevant for that specific project. Furthermore, derived products may contain modifications of platform functionality. Hence, in the studied setting, the platform is something *almost, but not quite, entirely unlike* a 150% model.

Figure 2 shows the four different relationships encountered between software components in the platform and the derived products. In this work, we are particularly interested in case 2, in which components are copied from the platform and then edited for specific use in a derived product. This case is primarily interesting from the co-evolution perspective since common and product-specific functionalities can both be present in software components. Changes to components in the platform might need to be propagated to the derived products. But it is no longer clear how those changes should be propagated because the common portions between platform and products are no longer identical nor are components in the products trimmed-down versions of those in the platform. As mentioned earlier, the absence of a well-defined relationship between the models complicates the co-evolution process, since it cannot rely on e.g. a conformance relation as in the co-evolution of metamodels and models.

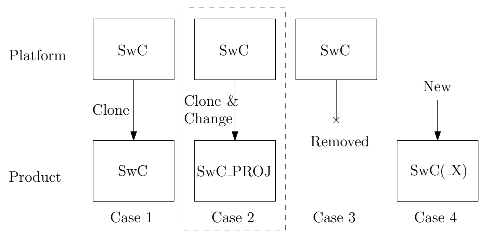


Figure 2: Encountered cases of relationships between software components in the derived products and their counterparts in the platform.

2.2 Making Software Changes

The systems under development are safety-critical, which means that all developed products require certification according to domain-specific industry safety standards as well as local and national regulations. Safety assessments are time-intensive and costly. Consequently, a large development effort lies in testing and certifying the software components. One of the improvements to this process was the introduction of a software product line because certification of products can then be based on existing certifications of the platform. Within the current development setting, upon an update to a software component in the platform, a review is needed for each of the products using that component, to assess if they can and should be updated as well.

A schematic co-evolution scenario is illustrated in Figure 3. Let N denote a model of a platform component and N_1 a cloned and subsequently modified copy of N in a derived product. Now consider an evolution of the platform where N changes. The evolved model is denoted as N^*. Since N_1 is based on N and probably contains cloned parts of N that are now updated, we may need to co-evolve the model of the product component too, thereby creating N_1^*.

Assessing the impact of a change in the platform on the products requires knowledge of how products are updated after that change. First, it is checked which products use the updated platform component. Then, for each of those products, an expert assesses the need for updating it (bug fixes are more likely to be propagated to the products than new functionality). Currently, this assessment process is completely manual, but given the scale and growth of the product line, this has become overwhelming, tending to infeasible. The platform contains about 180 software components, each of which is implemented as Simulink models and is associated with its own test harnesses and test cases. The mean number of top-level blocks in such a model is 40, with a standard deviation of 24. In the near future, the team will be working on six products derived from the platform. Given that an experienced engineer can check approximately one component every ten minutes, a new release of the platform causes a workload of one month for these reviews only. Secondly, upon a decision to propagate the changes to the product, the engineer manually updates the software components in the product. After the changes are incorporated, the product needs to be retested. Since changes might include new functionality, retesting may also require the development of new test cases or modifications to existing ones. In some cases, the updated test cases can be taken directly from the platform, but this cannot be guaranteed in a general case, due to the informal relationship between platform and product described earlier. Although the changes are performed manually, most effort lies in fact in reviewing the tests to make sure that they are still relevant and complete after a change to the software component and, in case they are not, to update them.

The overall goal of our research is to provide means by which the review process can be reduced and the number of tests needed when a component is updated can be limited. In our work, we focus on the first part of the review process, deciding whether platform changes should be propagated to products or not. The context of our work is thus the assessment of the impact of changes in the platform on software components and test cases in the derived products.

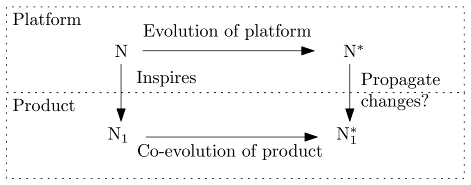


Figure 3: Schematic overview of the relation between platform evolution and product co-evolution.

2.3 Co-evolution in the Product Line

One way of considering the co-evolution problem is by seeing it as a three-way-merge. In three-way merging, three revisions of an artifact are merged into one. This is commonly used to version development artifacts that are collaboratively developed, in cases where local changes must be merged into an artifact that has in the meanwhile also been changed by someone else. One of the three revisions is considered the "base" artifact, from which the others are derived. These are usually named "theirs", for the remote revision, and "mine" for the local revision. In our case, we could consider the original platform (N) as the "base" model, the evolved platform (N^*) as "theirs", and the product before co-evolution (N_1) as "mine". A three-way merge is then expected to yield the co-evolved product as the "target" (N_1^*).

Three-way merging is a conceptually valid approach of ending up with the co-evolved product [26]. Note however that this merge implies that the target contains all changes as made in the platform. In our scenario, we do not necessarily want to achieve that. Rather, the engineers should be in control of the co-evolution and choose which changes should or should not be propagated to the products. Furthermore, although Simulink contains native support for three-way merging, we found that in some cases using this support results in semantically incorrect or irrelevant target models. An example of this issue is shown in Figure 4, where a base model was altered into two semantically identical, but syntactically different, ways. The resulting three-way merge is nonsensical since the input of each of the two *AND* gates depends on the output of the other.

Typically the expected output from this type of input would be a merge conflict, which then requires manual resolution, in this example however, the merge is performed without complaints.

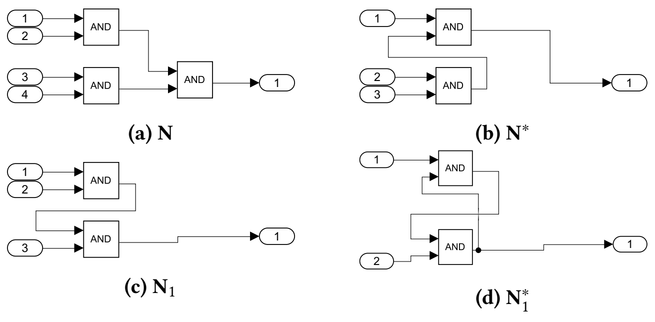


Figure 4: Example of a three-way-merge of Simulink models with a semantically incorrect result without raising merge conflicts. The arrangement of the four models corresponds to Figure 3.

Given that we do not always want to propagate all changes, and, in case of a merge, a manual resolution of merge conflicts may be required anyway, we might consider a different approach. We hypothesize that instead of merging models at the level of individual boxes and lines, the level of abstraction can be raised to that of portions of functionality contained in fragments of models, akin to the idea of feature-based product line evolution [19], but without the creation of a feature model. Within this scheme, an expert can choose to propagate changes to a model fragment encompassing a certain functionality, when this fragment is reused in different products. Assuming this support on the level of portions of functionality, it would be easier to create the co-evolved product, automatically assess the impact of the change, and thus aid in the assessment of the need for propagating it to multiple derived products.

In conclusion, we study the co-evolution of a platform and the products derived from the platform. Our primary focus lies in assisting engineers in assessing change propagation at the level of model fragments rather than low-level blocks, leading to the simplified creation of the evolved derived product (N_1^*).

3 APPROACH

The approach outlined in this section aids domain experts in co-evolving derived products upon evolution in the platform, by accepting or rejecting changes at the level of model fragments. Our approach is based on the observation that the product line is created by a "clone-and-own" approach, i.e., products are derived by copying the platform and then customizing them. Most software components in the products are expected to be a trimmed-down version of the corresponding software components in the platform, although the customization can also entail additions or modifications. Hence, the product software components are typically expected to contain reduced parts of functionality as compared to the platform.

We consider the schematic example models in Figure 5. The models on the top row (Figures 5a and 5b) represent the platform before and after evolution. The models in the first column of the two bottom rows (Figures 5c and 5e) represent two derived products before co-evolution. Within these figures, the shapes represent model elements. Hence, the example evolution from N to N^* shows an addition, a deletion, and a modification of a model element.

The overall approach consists of the four steps listed in Table 1. The last three steps represent a common software engineering pattern of packing, then doing something on the packed thing, and then unpacking again. This section further details each step using schematic example models. Section 4 details the implementation and feasibility study of our approach on a Simulink model that provides a realistic[1] representation of a typical model from the studied development setting.

Table 1: High-level overview of the steps in our approach, elaborated throughout Section 3 and exemplified throughout Section 4.

Step 1	Detect clones between platform and derived products.
Step 2	Replace clones with subsystem references.
Step 3	Evolve the referenced subsystem.
Step 4	For each reference, revert or expand the subsystem.

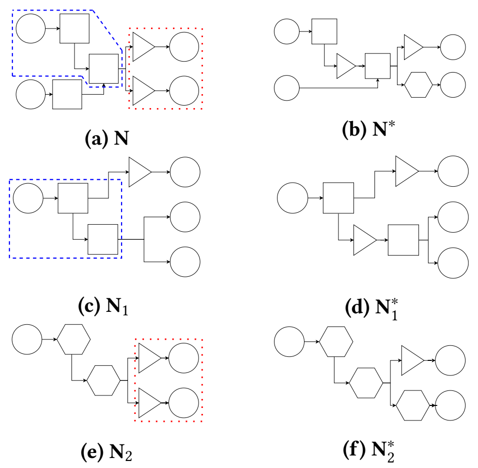
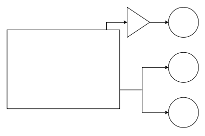
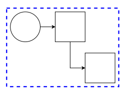


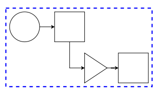
Figure 5: Schematic example of models in a platform (N), its evolved version (N^*), and two derived products before (N_1 and N_2) and after (N_1^* and N_2^*) co-evolution. Clones between the platform and derived products are indicated by the dashed and dotted regions in the models before evolution (N, N_1, and N_2).

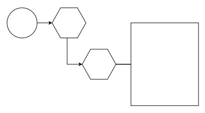
Step 1. The first step of the approach is to find common functionality shared between product and platform. This step can be skipped when traceability information tracking reuse already exists but is needed in our case since no explicit knowledge of reused functionality is present in the artifacts. In our setting, we expect the origin of common functionality between different files to be through copying. Therefore, we start by looking for exact clones

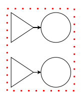
[1]Note that, for obvious reasons related to IP, we could not provide the "real" industrial model in the paper.


(a) N_1 with cloned fragment replaced by a subsystem reference.


(b) Library subsystem as referenced by N_1, before evolution.


(c) Library subsystem as referenced by N_1, after evolution.


(d) N_2 with cloned fragment replaced by a subsystem reference.


(e) Library subsystem as referenced by N_2, before evolution.

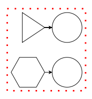

(f) Library subsystem as referenced by N_2, after evolution.

Figure 6: Cloned functionality is replaced with subsystem references (step 2). Then changes are applied in those referenced subsystems (step 3).

of model portions between platform and products, although differences in layout are acceptable as long as they do not change the semantics. The clones across the three models before evolution are illustrated by dashed lines in Figure 5, for N, N_1, and N_2 in Figures 5a, 5c, and 5e respectively. If desired, these results can be stored for future instances of co-evolution, although it should be noted that the derived products may evolve individually in the meantime and in that process break the links that are created in this step.

Step 2. The second step comprises of "packing" each of the cloned model fragments in separate subsystems. Crucially, these newly created subsystems are stored in a common library. In each derived product containing the cloned model fragment, that fragment is replaced by a reference to the created subsystem. The results of this step on the products N_1 and N_2 are shown in Figures 6a and 6d, respectively. Figures 6b and 6e show the subsystem as created in the common library and referenced by N_1 and N_2, respectively.

Step 3. Now, in all derived products, the cloned fragments have been replaced with a reference to the library subsystem containing that functionality. The third step applies the evolution of the platform to that library subsystem and, by these means, the change is automatically propagated to all derived products. To perform this step, we need to know the following four things:

(a) What evolution happened in the platform;
(b) Which cloned fragments are affected;
(c) How we can evolve the library subsystems;
(d) For each subsystem reference, whether the change should be accepted or discarded.

The first input we require is to know what evolution happened in the platform between its closest previous release and the current release. This can be obtained using standard model differencing provided by version control systems. Although notoriously difficult for graphical models, Simulink provides effective support for graphical model differencing. Note that the products, between their derivation from the platform and the new platform release, may have evolved on their own too. Therefore, when detecting clones, we consider the latest version of the derived product, rather than the version at the time of branching off from the platform.

Given an evolution of the platform, we need to determine which reused fragments are affected. In the schematic diagrams, we simply draw boxes around cloned fragments (as in Figure 5), providing clear clone borders. Naturally, this kind of meta-information is not present in the output of the clone detection step, nor it can be expected to be part of other traceability mechanisms. Nevertheless, it is important to assign borders to cloned fragments, since we want to limit the applied changes in this step to within cloned fragments, thereby allowing the engineers to adopt the changed fragment as a whole (as in Figure 6). Moreover, limiting changes to within fragment borders prevents problems when later (in step 4) "unpacking" the subsystem references again to their constituent blocks. Therefore, we define clone borders by the components within it, i.e., after a change, an added component shall be considered to be within the clone border when it is only connected to components already in the clone. In case of multiple new components, the same definition applies recursively. Modified components are considered within the clone scope if and only if they were in the clone scope before the evolution. Note that this step might result in cases where the change cannot be applied completely since it happens across the boundaries of the cloned fragment. We do not provide support for these specific cases, which are expected to be very few, given the nature of the clones in the studied setting.

Once we know which cloned fragments have changed and how they have changed, we want to apply the corresponding evolution to the library subsystems. For each clone, we can perform the same evolution as it happened in the platform, based on the obtained difference between the current and previous platform releases. Conceptually, this is similar to applying a patch as obtained through a Git *diff*. This is challenging in general for graphical models and not supported in Simulink.

Figures 6c and 6f show the situation after the platform changes are applied to the library subsystems. Note that in this example each subsystem contains a single change and that one of the changes in the platform, that falls outside the clone boundaries, is not propagated to any subsystem.

After this step, we can "unpack" the subsystems again and obtain the co-evolved models. Before doing so, test cases and calculations of other model metrics can be executed; this can help the engineer to decide if this change shall be propagated to the product, or if it shall be discarded and thereby revert to the original subsystem. This assessment, labeled as step 3d, requires domain knowledge and is therefore always manual.

Step 4. In step four, we break the references to the common subsystem, making the subsystems, in each model instance, unique. Furthermore, we replace the subsystems with their content, thus restoring the models to the same structure as before the co-evolution (but with different contents). Instead of these unpacking actions, we might consider keeping the now uncovered traceability links between clones. We opt to unpack the subsystems back to their constituent blocks to keep allowing the derived products to evolve independently of the platform too. Figure 5 shows the evolved products N_1^* (Figure 5d) and N_2^* (Figure 5f). Note that, as expected, changes outside the cloned fragments are not propagated to the products.

4 FEASIBILITY STUDY

In the first part of this section, we describe the implementation[2] of each of the steps as outlined in Section 3. Individual steps are supported by fully automated means, except for step 3, which requires an engineer's decision. The different steps are implemented as prototypes, we have not yet created automation to combine the steps into a single executable script. Since the models at our industrial partner cannot be shared, the second part of this section describes the application of the implemented approach on a realistic (not real) model. The third subsection describes experiences from applying the approach to real industrial models.

4.1 Implementation

Step 1. In the studied setting, no traceability information indicating reuse of model fragments exists. However, due to the "clone-and-own" nature of the product line, we expected to find exact clones across models. To detect them, we considered three alternatives for clone detection. The first is a built-in Simulink feature that can detect similar subsystems within the same model. In our case, we found that the models are quite "flat", i.e., they typically do not contain subsystems. Furthermore, we aim to find clones across different models, not within one single model.

After that, we considered the SIMONE Simulink code cloning tool [1]. SIMONE can be configured to look for similar subsystems or similar models. Its strength is the ability to detect near-miss clones. In this use case, we expect exact clones at block-level, which is why we eventually opted for the *ConQAT* [9] tool. ConQAT is used in several research works that consider Simulink clones, e.g., to detect anti-patterns [29], or to compare performances of cloning detection tools [1]. It can find the exact clones of model fragments across models by considering the Simulink models as graphs and matching identical sub-graphs. The tool relies on the textual storage format of Simulink models, so it requires input models to be saved as `.mdl`. For practical use, the main challenge is to configure the correct minimum size of to-be-detected clones in order to detect meaningful fragments. Through experimenting on our set of industrial models, we found out that a minimum of 5 blocks yields reasonable results.

Step 2. Step 1 yields cloned model fragments across several models (the platform and one or more derived products). In step 2, we want to convert these fragments into references to a common library subsystem. To do that, we first create a subsystem out of the cloned fragment. Executing the command `Simulink.BlockDiagram.createSubsystem(handles)` creates a subsystem from a list of handles of blocks making up a cloned fragment. This list is the result of step 1. To be able to be converted to a subsystem reference, Simulink requires a subsystem to be *atomic*.[3] This can be achieved by setting the parameter `TreatAsAtomicUnit` of a subsystem to `on`. After that, the subsystem can be replaced by a reference by executing the command

```
Simulink.SubSystem.convertToModelReference('ModelName
/SubsystemName', 'libref', 'ReplaceSubsystem', true).
```

Note that after these actions, a single cloned fragment in one of the models is replaced with a subsystem. However, in cases where a fragment is cloned across multiple derived products, an extra step is required to put a subsystem reference in each of them. First, a subsystem reference is created in each of the target models. In this scenario, we want to make all the subsystem references to point to the same library subsystem. This is achieved by changing the `ModelFile` parameter of subsystems to point to one library file containing the common subsystem.

Step 3. After step 2, all clones are replaced by subsystem references pointing to the same library subsystem. Consequently, any changes in that library subsystem during step 3 are automatically propagated to all models containing a reference to it. Now we need to apply the evolution that happened in the platform and within the clone to the subsystem library.

First, we find out what the change implies. In Simulink, the difference between two models can be obtained using either `visdiff(N, N$^*$)`, which creates a visual comparison and a report, or through `slxmlcomp.compare(N,N$^*$)`, which returns the difference in an `xmlcomp.Edits` object.

However, the latter is also mostly visual, since the object's main purpose is to allow the creation of a comparison report as created by the former command. Nevertheless, the `xmlcomp.Edits` object provides the roots of tree representations of both models and allows a programmatic traversal of them. This makes it possible to automatically look for all changed (parameter `Edited` set to `true`) or new blocks. The object will contain only those nodes that have changed, so all common ones that are unchanged are not mentioned. Using the same traversing technique, we can determine which clones are affected by the particular evolution in the platform and which changes are made to them. This is how the calculation of differences and their localization in code fragments could be implemented in the Simulink environment. However, Simulink does not provide features similar to Git, in which differences between files can be stored into a patch that can later be applied to the original file to obtain the changed one. Therefore, applying the changes is currently a manual process and consequently, there is not any notable added value in automating the remaining sub-steps. Upon the decision to accept the change, we continue with the final step, alternatively, the changes are discarded and the model is returned to the state before the changes made in step 2.

Step 4. Step 4 should break the links to the referenced subsystem and unpack the subsystems back to individual model components. We adopted a script from the Mathworks forums to break the link to the library subsystem and make the subsystem in the derived model unique [30]. After that, we converted subsystems back to their original components through `Simulink.BlockDiagram.expandSubsystem` with the subsystem that should be unpacked as an argument.

[2]All mentioned scripts and models are available in the following GitHub repository: https://github.com/RobbertJongeling/Simulink-PL-co-evo
[3]The execution of blocks in the atomic subsystem can not be interleaved with the execution of blocks outside it.

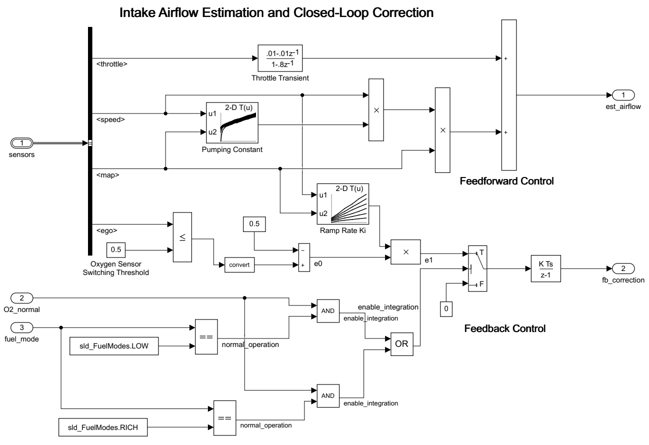


Figure 7: Realistic Simulink model as adapted from one of the Simulink examples. This model is considered the base platform model (N) in this Section.

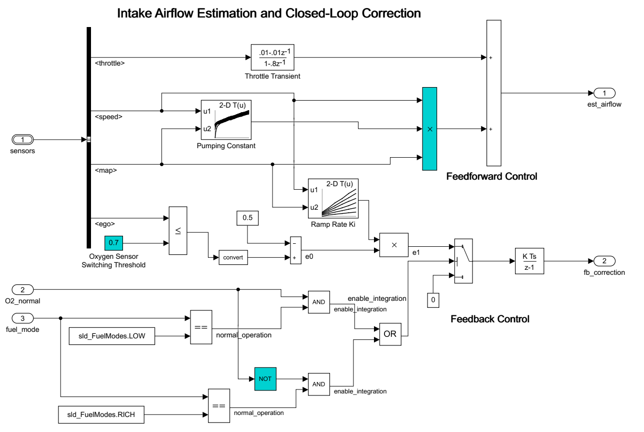


Figure 8: Evolved version of N from Figure 7, locations of changes indicated in cyan. This model is the evolved platform (N*) in this section.

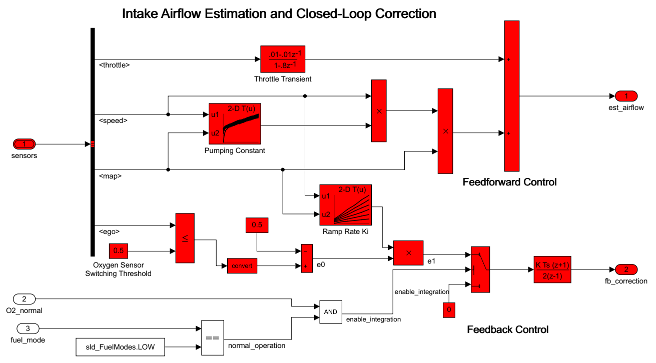


Figure 9: Model version derived from N from Figure 7. This model is the derived product (N_1) in this section. Highlighted in red are the blocks that form the fragment of N_1 that is cloned from N.

4.2 Demonstration

We illustrate the implementation by applying it to a public Simulink example model. In selecting the example, we considered models that are as realistic as possible in the sense that they 1) consider pieces of control software that could be used to generate code, 2) have a size and complexity comparable to typical models encountered in the studied industrial setting, and 3) can have derived products and can be subject to additions, changes, and deletions of model elements. We consider as a model the `airflow_calc` subsystem within the `fuel_rate_control` subsystem within the `sldemo_fuelsys` example model [14]. We slightly modified that by changing the condition for enabling the switch from (O2_normal ∧ fuel_mode = LOW) to ((O2_normal ∧ fuel_mode = LOW) ∨ (O2_normal ∧ fuel_mode = HIGH)). The resulting base model (N) is shown in Figure 7.

To show a reasonable example of evolution, we made sure to consider different types of changes, both within and outside a cloned fragment. The evolution of the model contains three changes compared to the base.

(1) It fixes a bug by adding a negation in the switch condition, thus making it ((O2_normal ∧ fuel_mode = LOW) ∨ (¬ O2_normal ∧ fuel_mode = HIGH)).
(2) It refactors two multiplication blocks with 2 inputs each to a single multiplication block with 3 inputs, in the "Feedforward Control" part.
(3) It updates the constant value Oxygen Sensor Switching Threshold from 0.5 to 0.7.

The resulting evolved model (N*) is shown in Figure 8.

We consider one product derivation from the original platform model. The derived product model is, as typical in our setting, cloned from the platform model and then modified. Two changes were made after the cloning step:

(1) Part of the switch condition functionality was stripped, leaving only the original from the `sldemo\_fuelsys` example: (O2_normal ∧ fuel_mode = LOW).
(2) The integrator method used by the discrete integrator block has been changed from `Forward Euler` to `Trapezoidal`.

The resulting derived model (N_1) is shown in Figure 9.

Now we show the steps of our approach and how they could be applied to this use case.

Step 1. We ran one of the ConQAT pre-defined configurations that detect clones between Simulink models: simulink-analysis.cqr. As input, we used `.mdl` versions of the original model (N, Figure 7) and the variant (N_1, Figure 9), and we configured a minimum-clone-size of 5. In this example, a single large cloned fragment is identified between N and N_1, it is shown in red in Figure 9. Note that the integration method is a property of the discrete integrator block and since the clone tooling works on the level of blocks, the two integrator blocks are marked as clones.

As input for step 2, we require a list of handles of all blocks in the cloned fragment. ConQAT reports a list of block identifiers in the form of a Matlab script (`.m` file) that can be run to obtain the colorization as shown in Figure 9. It does this by setting, for each

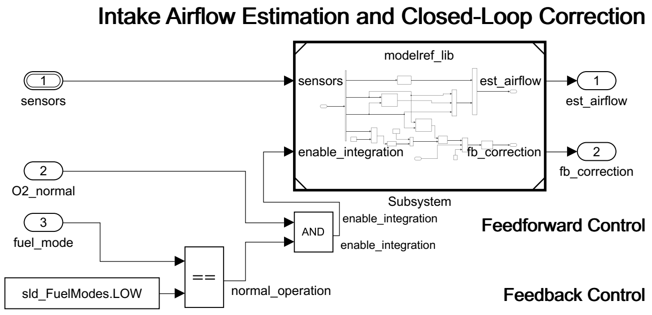


Figure 10: N_1 being prepared for co-evolution, clone is replaced by subsystem reference.

block in the model, the parameter BackgroundColor. To obtain a list of handles, we modified this output file to, instead of setting these parameters, getting the 'handle' parameter for all blocks that are part of the cloned fragment (which they are when their background color is set to anything else than white). So we take the generated simulink-analysis_matlab-colorm-file-writer_Clone_0.m file (and possibly the other files, if more clones were found), and then we run the script to get the block handles out.

Step 2. Now we have all the handles of the blocks in the subsystem. At this point we make a subsystem out of all the cloned blocks, we make it atomic and finally we convert it into a reference:

(1) `Simulink.BlockDiagram.createSubsystem(handles);`
(2) `set_param('intakeAirflow_var/Subsystem', 'TreatAsAtomicUnit', 'on');`
(3) `Simulink.SubSystem.convertToModelReference( 'intakeAirflow_var/Subsystem', 'modelref_lib', 'ReplaceSubsystem', true)`

After applying these steps, model N_1 contains a reference to the library subsystem as stored in `modelref\_lib`. The resulting model is shown in Figure 10

Step 3. The engineer shall now apply to the library subsystem the changes that are limited to the clone scope, as defined in Section 3. Following our rules, the new negation block is not part of the change to be applied, but the changed parameter and the multiplication refactoring are. The changes made in the library are immediately visible also in the variant since it contains a reference to it. Now that the changes appear in the variant, it is up to the domain expert to choose whether to accept or discard the changes made by the co-evolution mechanism. In this example, we assume that the changes are accepted, to continue with step 4.

Step 4. For breaking the link to the library subsystem, we run the aforementioned script: `Replace_MdlRef_SubSys( 'intakeAirflow_var')`. As a small note on an implementation detail, this script copies the model and breaks the link in the copy, but this is not essential to the functionality and not relevant for our approach. Now that the link is broken, we can unpack the subsystem again to its constituent blocks through `Simulink.BlockDiagram. expandSubsystem('intakeAirflow_var_new')`. We obtain the model as shown in Figure 11, which represents N_1^*, the co-evolved derived product.

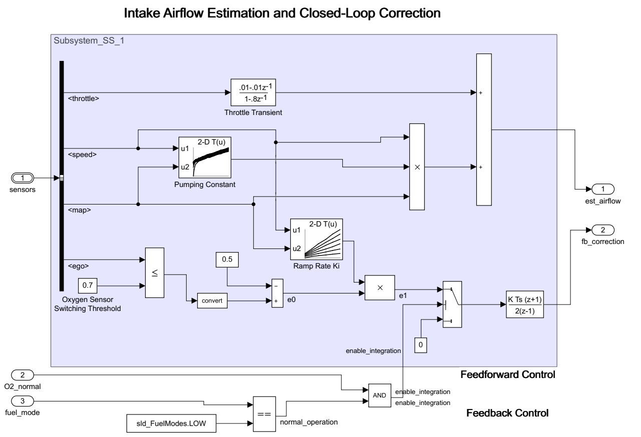


Figure 11: The result of the co-evolution approach: N_1^*. Note that the background is there just to clarify the image. It can be removed by setting the additional option 'CreateArea' to 'Off' (available from 2019a)

4.3 Experiences

We now describe the outcomes of applying our approach to real industrial models. We discuss in particular two co-evolution cases of software components (referred to as case A and case B) that are particularly interesting because they represent extreme cases in the described setting. The derived products are large reductions of the platform (From 80 to 20 in case A and from 74 to 15 top-level blocks in case B) and contain some changes.

Step 1. In case A, two clones were found, one consisting of eight blocks and another consisting of seven blocks. The platform contains two occurrences of a pattern of a few blocks; one occurrence is included in the clone. It is not clear why one is chosen over the other. Case B contains only a single clone, by the size of ten blocks.

In general, detecting clones using ConQAT worked fairly well for our use case, because we expect identical clones between platform and components. Nevertheless, it is still challenging to properly configure the tool (e.g., the minimum size of clones) to obtain desired results. Another potential hiccup is the requirement of the clone detection tool to provide input models in the textual `.mdl` format, which is not supported for all models, for example, those including internal test harnesses. We did not encounter this problem in our setting since all test harnesses had previously been exported.

An underlying assumption of our approach is that the software component models can be divided into meaningful fragments, representing small chunks of functionality, such as input handling, or a particular calculation. When applying the approach on a set of industrial models, we found that this assumption does not always hold. In particular, smaller models proved less suitable for our approach since the cloned fragments were either too small to be meaningful or not existing at all. Furthermore, for small models, a manual approach could be a better choice time-wise.

Step 2. In case A, since there are two clones, the step to create a subsystem from a cloned fragment has to be executed twice. This works fine, but one thing that could form a barrier for practical use is that the intermediate result can look messy since the creation of subsystems will cause other lines to rearrange and possibly get entangled.

Step 3. A considerable amount of changes in the revision history are non-functional. They are limited to e.g. updates to parameter names or comments. In case A, the change in the platform is a correction to the model for which an extra multiplication with a constant factor is added to a flow. In case B, also functionality is added to the model, but in contrast to case A, it is partially outside of the cloned fragment and is therefore not propagated by the approach, as described in Section 3.

Step 4. In case A, the unpacking functions correctly. It should be mentioned that for practical applications a hinder may be that the layout of the model after applying these steps may differ significantly from the layout before the steps. This holds even if the co-evolution is limited to the updates of parameter values.

Moreover, we have encountered a type of case in which our approach does not help the engineers much. One example concerns a refactoring impacting the entire model, rather than being localized to a specific model fragment. Examples of such refactoring efforts include: starting to monitor values on all output ports for testing purposes, externalizing a model's test harness, or upgrading the model to work with a new Simulink version. In these cases, our proposed approach would likely miss some occurrences, since they would occur outside cloned fragments.

5 DISCUSSION

We have shown our implementation of automatic support for co-evolution between models in an industrial setting of a software product line. The implementation can be integrated into the development process since it builds on the same Matlab platform as the Simulink models. Nevertheless, some points in our approach require further research, as identified throughout Section 4. For example, a possible scenario is that a single evolution affects several cloned fragments. In that case, propagating a few of them may not yield a meaningful model, for instance, because an extra value was introduced in one cloned fragment but not used in any other. This is somewhat related to the issue of changes across clone borders that cannot be propagated. In the end, there may remain cases for which the approach does not provide the desired result. However, thus far, these concerns have not impacted the application of the approach, because they did not occur in the studied industrial models. Overall, it is clear that the solution is not the holy grail and that several problems remain open.

In Section 4, we showed that the approach still requires human intervention for a few decision-making actions. Ideally, all steps not strictly requiring an engineer's decision would be automated, but the following implementation challenges have prevented us from doing that so far. To automate the application of changes in the library subsystem, it is required to be able to refer to the same block across different models. Handles are not carried over between models so it is required to rely on identifiers. However, most blocks within the models have no explicit name, but they rather get automatically assigned names (e.g. `LogicalOperator4`). This numbering system may completely change in the case of a derived product in which some modifications were made, hence identifiers are also not reliable to identify blocks across models. Even if this would be solved, there is still one action left to be done manually, which cannot be automated. More specifically, the engineer must decide whether to accept or discard the proposed co-evolution. By using our approach, the engineer gets access to more information to base that decision on, since the proposed co-evolution model can be tested and other model metrics derived from it.

When using the approach, an unexpected benefit was to get support for localizing reused fragments. There was no traceability for this in place except for an inconsistently followed naming convention at the level of model files that indicates if they are derived and modified after being branched off from the platform. Although requirements traceability is in place in the industrial setting, it specifically links requirements to Simulink models and test harnesses. The application of the clone detection tooling allowed instead for identification of reuse at sub-model level. In the studied setting, we encountered flat models, in contrast to the common practice in Simulink of creating hierarchies of nested subsystems. The approach is not dependent on the absence of subsystems, but the practical implementation is guided by it. Indeed, the typical absence of hierarchy in the studied setting and the expectation of finding exact model fragment clones have guided our choice of clone detection tool.

The studied problem exists in part because of the clone-and-own practice and the accompanying lack of traceability between re-used model fragments. As a way forward towards a product line, we mention two main directions. In the first, Step 4 of the approach could be omitted, which would allow the practice to migrate to a more systematic product line. In such a case, the identified clones might be lifted to a variant description model in pure::variants [5], which could then be used to generate different model variants. The second direction would be to have more support for product lining in the modelling language, such that clone-and-own may be avoided. Variability in Simulink can be managed through feature modelling, negative variability, or delta modelling [2]. In our setting, we would want to manage variability outside single models, given the strict regulations on their certification. Therefore, feature modelling and managing variability using pure::variants could be a good alternative in the studied setting, despite the required changes throughout the engineering process to fully benefit from this way of organizing reuse.

It is clear that the proposed approach is heavily guided by the used modelling tooling and practices in the studied setting. Nevertheless, we can imagine generalization to other modelling languages. At a very high level, the approach is not uniquely applicable to Simulink only, we can imagine finding clones between models and then replacing them with references in other languages too. However, this depends also on the storage format of models. For modelling tools with repository-based instead of file-based storage, variability may be addressed in a completely different way, making our approach less applicable. We finalize this discussion by reiterating that, in this work, we address an existing clone-and-own practice and aim to provide means to improve it, despite knowing that it is not the ideal product line practice.

6 RELATED WORK

Within MDE, co-evolution commonly refers to model-metamodel co-evolution [6]. In contrast, co-evolution in software product line engineering commonly refers to the parallel evolution of a feature model and software model. For example, recent work on co-evolution in model-based product lines enables the co-evolution of a feature model and model variants, through an approach combining the management of revisions of models through time and across variants [27]. In this work, we have considered a third option, the co-evolution of software models belonging to a platform and derived products in a model-based software product line. A key difference between existing co-evolution work and our work is that we cannot rely on the formal conformance relation between artifacts, as is typically done in operator or inference approaches to co-evolution in MDE [16]. Instead, in our case, the derived products are merely inspired by the platform, but there are no more guarantees about the relation between them.

We study the co-evolution of Simulink models between which fragments have been reused through exact copies. Alternatively, Rumpe et al. propose to assess reuse opportunities of Simulink model fragments by checking that the behaviors of different fragments are the same [25]. In their work, they also consider a software product line setting. They note, similar to us, that behaviorally identical fragments across models may be replaced with references to a single library fragment. Due to the expectation of encountering exact syntactic clones in the studied setting, we do not further study the identical behaviors of different models. Other work also considers co-evolution of model fragments [19], but in contrast to our work, they construct a feature model and consider the automatic creation of variants from it. In our case, the variants have already been created and must be co-evolved with the platform from which they are derived. A similar study checks the integrity of the co-evolution by utilizing notations from evolutionary biology that keep track of the variations between the platform and derived products [3]. Other work considers co-evolution of models and libraries [4]. Although that work seems closer to our studied setting, still it relies on a formal conformance relationship between the models and the libraries.

There have been other works considering variability in Simulink models. Dajsuren has proposed to manage clones and their variants outside of models, through configuring clones using a textual language [7, Chapter 7]. The approach allows for reuse of subsystems across different models while keeping the clone management separate from the main model, thus preventing overloaded models from growing too large and becoming unreadable. Another proposal to prevent overly overloaded models is to define model variants using the set of operations that are required to obtain the variant from the base model, so-called delta-modeling [12]. In this work, we have a different starting point, since we already encounter an existing clone-and-own product line. Neither of the aforementioned approaches specifically considers co-evolution of the cloned fragments. Other work has specifically explored three-way-merging as a solution to co-evolution in software product lines [26]. As we have argued, there is an inevitable need for human intervention in those cases and as we then hypothesize, resolving conflicts would be easier when the engineer can reason about them at the level of functionalities rather than individual blocks.

There has also been some work towards impact analysis [21] and test evolution [22] for Simulink models in an industrial setting. Since the proposed approaches and implementations in those works are tightly integrated with the industrial setting in which they are developed, their results are not so easily generalized. The impact analysis work assesses the impact of changes in a model on their tests by forward and backward traversal of the flow in the model, until reaching input and output ports. All tests involving those inputs or outputs are then said to be impacted. The test evolution work builds further on that test identification work to generate test harnesses in case the evolution requires the creation of a new one. Hence, the work limits consideration of co-evolution to tests. This is one of the aspects we aim to consider in our future work, whereas in this paper we have focused on co-evolution of software models.

Related work on clone management [15] emphasizes its need but focuses on code-based software representations. Within code clone analysis, evolution of code clones is often studied with the aim of visualizing it [23]. Moreover, those studies have reported disagreeing findings from empirical studies [17]. In addition to these code-based clone evolution studies, other works have considered graph-based (Simulink) models as well. ModelCD is a clone detection tool based on ConQAT that can detect exact and near-miss clones across Simulink models [18]. The evolution of cloned Simulink fragments has been studied before too [28]. In that work, the authors focus on identifying clones across revisions of Simulink models to study their evolution.

7 CONCLUSION

We have presented our approach for the co-evolution of Simulink models in a model-based product line. Co-evolution is considered at the level of model fragments that are cloned across a platform and derived products. In this work, we aimed to provide support for the process of assessing whether or not a change should be propagated. The feasibility of our approach is shown on a realistic example model. We showed that our approach yields correct results and that automated support aids in the decision-making process.

To improve the practical applicability of this work, we plan to extend our approach to include "what-if" analysis on the generated co-evolved products. This kind of analysis aids engineers in assessing whether to accept or discard a change and ascertaining how associated test cases should be updated given a change to a model. We will then build upon that for another planned future work, where we plan to provide more support for semi-automatically co-evolving the test cases as well. This will contribute to one of our main goals, i.e. reducing the effort of review and test upon platform evolution. Another interesting future direction is to consider alternatives for the code clone detection step. For that, we are looking into combining our work with other research efforts carried out at our partner company which focus on the automatic identification of potential model fragment reuse based on requirements similarity.

REFERENCES

[1] Manar H Alalfi, James R Cordy, Thomas R Dean, Matthew Stephan, and Andrew Stevenson. 2012. Models are code too: Near-miss clone detection for Simulink models. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 295–304. https://doi.org/10.1109/ICSM.2012.6405285

[2] Aitor Arrieta, Goiuria Sagardui, and Leire Etxeberria. 2014. A comparative on variability modelling and management approach in simulink for embedded systems. *V Jornadas de Computación Empotrada, ser. JCE* 26-33 (2014).

[3] Anissa Benlarabi, Bouchra El Asri, and Amal Khtira. 2014. A co-evolution model for software product lines: An approach based on evolutionary trees. In *2014 Second World Conference on Complex Systems (WCCS)*. IEEE, 140–145. https://doi.org/10.1109/ICoCS.2014.7060991

[4] Luca Berardinelli, Stefan Biffl, Emanuel Maetzler, Tanja Mayerhofer, and Manuel Wimmer. 2015. Model-based co-evolution of production systems and their libraries with AutomationML. In *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE, 1–8. https://doi.org/10.1109/ETFA.2015.7301483

[5] Danilo Beuche. 2003. *Variant management with pure:: variants*. Technical Report. Technical report, pure-systems GmbH, 2003. http://www. pure-systems. com.

[6] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2008. Automating co-evolution in model-driven engineering. In *2008 12th International IEEE Enterprise Distributed Object Computing Conference*. IEEE, 222–231. https://doi.org/10.1109/EDOC.2008.44

[7] Yanjindulam Dajsuren. 2015. *On the design of an architecture framework and quality evaluation for automotive software systems*. Ph.D. Dissertation.

[8] Charles Darwin. 1909. *The origin of species*. PF Collier & son New York.

[9] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. 2008. Clone detection in automotive model-based development. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 603–612. https://doi.org/10.1145/1368088.1368172

[10] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2011. What is needed for managing co-evolution in MDE?. In *Proceedings of the 2nd International Workshop on Model Comparison in Practice*. 30–38. https://doi.org/10.1145/2000410.2000416

[11] 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. https://doi.org/10.1109/ICSME.2014.61

[12] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. 2013. First-class variability modeling in Matlab/Simulink. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. 1–8. https://doi.org/10.1145/2430502.2430508

[13] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. 2016. Model-Based Engineering in the Embedded Systems Domain: an Industrial Survey on the State-of-Practice. *Software & Systems Modeling* 17, 1 (2016), 91–113. https://doi.org/10.1007/s10270-016-0523-3

[14] MathWorks. 2017. *Modeling a Fault-Tolerant Fuel Control System*. Retrieved May 24, 2020 from https://mathworks.com/help/simulink/slref/modeling-a-fault-tolerant-fuel-control-system.html

[15] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H Pham, Jafar Al-Kofahi, and Tien N Nguyen. 2011. Clone management for evolving software. *IEEE transactions on software engineering* 38, 5 (2011), 1008–1026. https://doi.org/10.1109/TSE.2011.90

[16] Richard F Paige, Nicholas Matragkas, and Louis M Rose. 2016. Evolving models in model-driven engineering: State-of-the-art and future challenges. *Journal of Systems and Software* 111 (2016), 272–280. https://doi.org/10.1016/j.jss.2015.08.047

[17] Jeremy R Pate, Robert Tairas, and Nicholas A Kraft. 2013. Clone evolution: a systematic review. *Journal of software: Evolution and Process* 25, 3 (2013), 261–283. https://doi.org/10.1002/smr.579

[18] Nam H Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M Al-Kofahi, and Tien N Nguyen. 2009. Complete and accurate clone detection in graph-based models. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 276–286. https://doi.org/10.1109/ICSE.2009.5070528

[19] Andreas Pleuss, Goetz Botterweck, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. 2012. Model-driven support for product line evolution on feature level. *Journal of Systems and Software* 85, 10 (2012), 2261–2274. https://doi.org/10.1016/j.jss.2011.08.008

[20] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.

[21] Eric J Rapos and James R Cordy. 2017. SimPact: Impact analysis for simulink models. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 489–493. https://doi.org/10.1109/ICSME.2017.21

[22] Eric J Rapos and James R Cordy. 2018. SimEvo: A Toolset for Simulink Test Evolution & Maintenance. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 410–415. https://doi.org/10.1109/ICST.2018.00049

[23] Chanchal K Roy, Minhaz F Zibran, and Rainer Koschke. 2014. The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 18–33. https://doi.org/10.1109/CSMR-WCRE.2014.6747168

[24] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2013. Managing cloned variants: a framework and experience. In *Proceedings of the 17th International Software Product Line Conference*. 101–110. https://doi.org/10.1145/2491627.2491644

[25] Bernhard Rumpe, Christoph Schulze, Michael Von Wenckstern, Jan Oliver Ringert, and Peter Manhart. 2015. Behavioral compatibility of simulink models for product line maintenance and evolution. In *Proceedings of the 19th International Conference on Software Product Line*. 141–150. https://doi.org/10.1145/2791060.2791077

[26] Sandro Schulze, Michael Schulze, Uwe Ryssel, and Christoph Seidl. 2016. Aligning coevolving artifacts between software product lines and products. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. 9–16. https://doi.org/10.1145/2866614.2866616

[27] Felix Schwägerl and Bernhard Westfechtel. 2019. Integrated revision and variation control for evolving model-driven software product lines. *Software and Systems Modeling* 18, 6 (2019), 3373–3420. https://doi.org/10.1007/s10270-019-00722-3

[28] Matthew Stephan, Manar H Alalfi, James R Cordy, and Andrew Stevenson. 2013. Evolution of Model Clones in Simulink. In *ME@ MoDELS*. Citeseer, 40–49.

[29] Matthew Stephan and James R Cordy. 2015. Identification of Simulink model antipattern instances using model clone detection. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 276–285. https://doi.org/10.1109/MODELS.2015.7338258

[30] MathWorks Support Team. 2013. *How can I replace my Model Reference block with a Subsystem in Simulink?* Retrieved May 24, 2020 from https://www.mathworks.com/matlabcentral/answers/98940-how-can-i-replace-my-model-reference-block-with-a-subsystem-in-simulink