

Towards Continuous Round-trip Engineering of Architectural Performance Models

1st Alp Toraç Genç*, 2nd Martin Armbruster^{✉*}, 3rd Manar Mazkatli^{✉*}, and 4th Anne Kozirolek^{✉*}

*KASTEL - Institute of Information Security and Dependability

Karlsruhe Institute of Technology

Karlsruhe, Germany

ufyzm@student.kit.edu, martin.armbruster@kit.edu, manar.mazkatli@kit.edu, kozirolek@kit.edu

© 2026 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Publisher's Version: A. T. Genç, M. Armbruster, M. Mazkatli and A. Kozirolek, "Towards Continuous Round-Trip Engineering of Architectural Performance Models," 2026 *IEEE 23rd International Conference on Software Architecture Companion (ICSA-C)*, Amsterdam, Netherlands, 2026, pp. 409-416, doi: 10.1109/ICSA-C68850.2026.00079.

<https://doi.org/10.1109/ICSA-C68850.2026.00079>.

Abstract—Architectural performance models (APMs) achieve a high-level view on source code, enriched with performance-related annotations. This allows performance predictions to explore design alternatives. As a result, the APM can be adapted to gained insights. Since iterative software development practices (e.g., DevOps) embrace frequent code changes, keeping APMs and their source code consistent can become challenging and laborious. Thus, we want to investigate the continuous round-trip engineering between APMs and source code in DevOps. In this paper, we report on our current work in progress in this direction, where we build upon the Continuous Integration of architectural Performance Models approach, which handles the code-to-APM direction. In this paper, we present an approach for the direction APM to code, which obtains changes on an APM, preprocesses them, and (semi-)automatically propagates them via incremental, rule-based model transformations to the source code. We prototypically implemented parts of the approach and provide an initial evaluation. In the future, we plan to extend the implementation and evaluation and to investigate the role of LLMs.

Index Terms—software architecture model, consistency preservation, performance model, round-trip engineering.

I. INTRODUCTION

The application of model-driven engineering in software projects can accelerate their development [1]. In particular, software architecture models can help to consolidate ideas and concepts by abstracting from implementation details [2]. In addition, if they include quality-related elements, they allow analyzing different quality attributes of a software system by predicting or simulating them [3]. For example, architectural

performance models (APMs) enable performance predictions and the exploration of design alternatives (e.g., different database technologies). As a result, the software architecture model can be adapted according to the insights gained from the performance or other quality predictions (e.g., which database technology to use) and to guide the implementation in the source code.

Especially in iterative development processes and practices such as DevOps, where software development focuses on fast release cycles and fast feedback [4], manual adaptations and synchronization of APMs and source code can hinder the APMs' application as manual work on them require time and effort. Resulting architectural drift leads to maintenance issues and quality degradation [5]. Thus, automated processes, reflected in the form of, for instance, round-trip engineering [6], can improve and support the synchronization of APMs and source code and, thus, can avoid architectural drift. To the best of our knowledge, however, there is currently no approach which enables round-trip engineering for APMs in DevOps processes.

Therefore, in this paper, we report on our current work in progress towards continuous round-trip engineering for APMs. In the end, we want to answer the following research question: How can APMs be round-trip engineered in DevOps? For the direction of source code to APMs, we build upon the Continuous Integration of architectural Performance Model (CIPM) approach by Mazkatli et al. [7]. It continuously and automatically preserves the consistency from source code to an APM in a proposed model-based DevOps pipeline. Nevertheless, the CIPM approach does currently not support the propagation of APM changes to source code. This leads to the following research question, on which this paper specifically focuses: How can APM changes be propagated to source code in DevOps? Based on this question, the contribution of this paper is an approach and initial evaluation for propagating APM changes to source code. The approach preprocesses APM changes before executing incremental, rule-based model transformations to source code. We developed a prototypical implementation for parts of the approach as an extension to CIPM.

The remaining paper is structured as follows: in section II, we introduce the relevant foundations for our approach. Then, we describe a running example in section III and the current

state of our approach in section IV, followed by an initial evaluation in section V. After covering related work in section VI, we conclude the paper in section VII with our next steps and future work.

II. FOUNDATIONS

Before describing our approach, we briefly introduce essential concepts in the following.

A. Architectural Performance Models

According to Gorsler et al. [8], *APMs* "describe the software architecture, the deployment, and are annotated with performance-relevant aspects of the software system" [8], enabling performance predictions at the architecture level. For our approach, we consider the *Palladio Component Model (PCM)* [3] as a concrete APM. The PCM is a framework to model component-based software architectures and predict their performance. It provides a *repository model* for software components and their behavioral aspects, system model, composing the components to a software system, model of resource allocations for the system, and system usage models. In the repository model, developers can specify components, interfaces with service signatures, which are provided or required by components, and data types for the service signatures.

B. The extended Java Model Parser and Printer

The *extended Java Model Parser and Printer (JaMoPP)* [9], [10] provides an explicit metamodel to represent Java source code (up to Java 15) one-to-one in a Java source code model. Additionally, it contains a parser for generating models from code and a printer to write code from a model.

C. Continuous Integration of architectural Performance Models

The *CIPM approach* by Mazkatli et al. [7] aims to automatically preserve the consistency between source code changes and APMs during software development, as well as adaptive changes and APMs during software operation, enabling accurate performance predictions. Therefore, the CIPM approach defines an automated model-based DevOps pipeline, which integrates related activities. Starting with a new commit in a version control repository, CIPM derives the changes between the previous and new commit. Then, these changes are applied to a source code model, acting as an intermediate model, from which the changes are incrementally propagated to an APM. As a result, the consistency between the source code and APM is preserved. Additionally, CIPM updates an *instrumentation model (IM)*, which captures changed source code parts for adaptive instrumentation. The instrumented source code is monitored in a test environment to calibrate the performance parameters in the APM and to validate the accuracy of the performance predictions. The CIPM approach mainly considers Java source code, using the extended JaMoPP, and PCM as APM. CIPM builds upon the VITRUVIUS approach [11], which is a framework for preserving the consistency between different models in view-based development. It currently operates on *atomic model changes*, which capture elementary

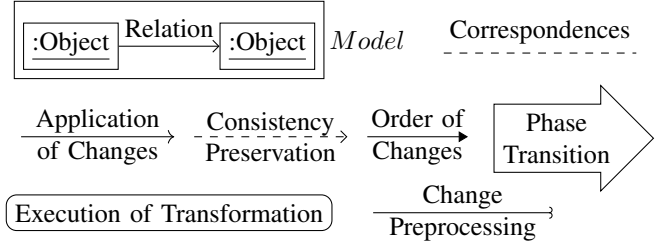
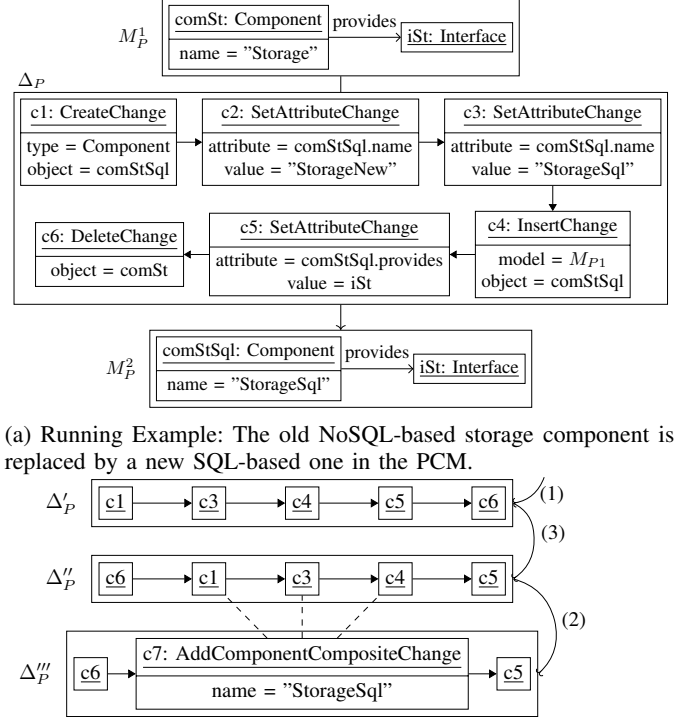


Fig. 1: Legend for Figure 2, Figure 3, and Figure 4.



(a) Running Example: The old NoSQL-based storage component is replaced by a new SQL-based one in the PCM.
(b) Example steps for the change preprocessing. Details for changes c1 to c6 are omitted for clarity.

Fig. 2: Example for obtaining and preprocessing changes.

operations on model elements that cannot be decomposed further. In contrast, *composite model changes* consist of atomic or other composite changes.

III. RUNNING EXAMPLE

To illustrate our approach, we use TEAMMATES [12] as a running example throughout the description of the approach. TEAMMATES is a real-world, cloud-based tool with a Java-based backend for feedback management for students and instructors [12]. Between version 8.0.0 and 9.0.0-beta.6, the old NoSQL-based storage component was replaced by a new SQL-based one. A potential modeling of this scenario in the PCM is shown in Figure 2a. There, the PCM model M_P^1 contains the old storage component. A set of changes $\{c1, \dots, c6\} = \Delta_P$ describes how the component is replaced. By applying this set on M_P^1 , the new model M_P^2 is generated, in which the old component is replaced by the new one.

IV. APPROACH

In the following, we describe our current approach and state for an incremental change-driven consistency preservation between APMs and source code, extending CIPM towards round-trip engineering. While our approach should remain applicable to different programming languages and APMs, we consider the PCM as concrete APM, the extended JaMoPP as source code model, and, consequently, Java as programming language.

Within our approach, we differentiate between two roles. The first one is the *software architect*, which modifies the APMs to, for example, adapt the software architecture to the results of performance predictions. The second role is the *software developer*, which is responsible for reviewing source code changes generated in response to APM changes and implementing remaining source code parts. We assume for now that each role only works on their respective model and the models are not changed simultaneously (i.e., if one role modifies their model, the other role waits until those modifications are propagated).

Our approach starts by obtaining APM changes (subsection IV-A), which are preprocessed (subsection IV-B) before they are propagated to the source code model (subsection IV-C). During the propagation, ambiguous APM changes can lead to manual user interactions (subsection IV-D) to resolve them. From the updated source code model, the actual source code is generated (subsection IV-E). We prototypically implemented parts of the approach (subsection IV-F).

A. Obtaining APM Changes

In the first step, we obtain the changes on the APM. There are two possibilities [11]: the changes can be either recorded in an editor or derived by comparing the updated APM with the older one. Orthogonally, the resulting change sequence can consist of completely atomic changes or a mixture of composite and atomic ones. In our approach, we assume that the software architect records their modifications to the APM in the form of an atomic APM change sequence, being independent of editor-specific composite changes and the support for recording composite changes. An atomic APM change sequence implicitly and partially captures the architect's intent, which we can use later to improve the APM change propagation. Our approach still supports atomic APM change sequences, derived from comparing an updated and its corresponding older APM via state-based model differencing [13]. In our running example, the atomic changes in Δ_P (cf., Figure 2a) were recorded.

B. APM Change Preprocessing

Since each individual atomic change represents an elementary operation on a model element, using only atomic APM changes may lead to contextual information loss. Moreover, recorded changes can capture all operations an architect performed on the APM, including changes that nullify each other. Therefore, in the next step, we preprocess the atomic APM changes before changing the source code model.

Consequently, we analyze and adjust the APM change sequence to improve and ease its propagation to the source code model. We iteratively search for specific patterns within the APM change sequence and transform them accordingly. These transformations include: (1) removing changes that are redundant or nullify each other, (2) replacing atomic changes with composite ones, and (3) reordering changes. Transformations (1) let us account for APM changes that have no effect on the resulting APM so that they do not need to be propagated to the source code model. By performing transformations (2), we aim to recoup the architect's intent and account for it during the APM change propagation, which can help to adjust the source code model better to reflect the APM. Transformations (3) allow us to shape the APM change sequence in ways that facilitate the APM change propagation. For these transformations, existing approaches can be used. This includes, for example, semantic lifting [14], heuristics, or LLMs.

In Figure 2b, we provide an example on how the change set Δ_P from our running example could be preprocessed. In the first step $\Delta_P \rightarrow \Delta'_P$, transformation (1) removes change c_2 since it is nullified by change c_3 . In the next step $\Delta'_P \rightarrow \Delta''_P$, transformation (3) selects change c_6 as the first change since it keeps the APM smaller for the remaining changes. Lastly, for $\Delta''_P \rightarrow \Delta'''_P$, transformation (2) aggregates changes c_1 , c_3 , and c_4 into a composite change c_7 , which represents the addition of a new component. In the following sections, we consider Δ'''_P as the change set, on which the running example operates.

C. Consistency Preservation between APM and Source Code

After preprocessing the APM changes, we propagate them to the source code model, using the VITRUVIUS approach [11]. The changes trigger the execution of incremental, rule-based model transformations, which preserve the consistency between the APM and source code model. During the change propagation, we maintain correspondences between APM and their related source code model elements. As a result, we keep track of how APM elements are represented in the source code model and vice versa. This allows us to determine the source code model elements that APM changes affect. In addition, we tag correspondences to embed further information, which is neither present in one of the models nor derivable from them (e.g., user input or configuration details).

For the APM to source code model transformations, we define rules, which describe how to adapt the source code model for certain APM changes. While defining these transformations, we can encounter ambiguous APM changes, for which we cannot fully automatically determine how to adapt the source code model, mainly due to the abstract nature of the APM. In such cases, we need to either ask the software architect for clarification or leave a task for the software developer. Since our approach aims to ease the consistency maintenance, we are facing a trade-off between comprehensiveness and automaticity of the APM change propagation: As we propagate more APM changes and change types, we may need to ask the software architect more frequently. Furthermore, particular

changes (e.g., changes in behavioral model parts) could require to ask the software architect about low-level implementation details, which should only concern software developers. Thus, in the interest of keeping both roles separated, interactions with the software architect should be limited to APM changes that do not target statement-level details in the source code (model).

For concrete transformations, we provide an initial, non-exhaustive set for transforming PCM changes to the Java model. Here, we focus on PCM elements that are relevant for source code. This means that we exclude PCM elements only embedding quality attribute-related annotations. Since we build upon the CIPM approach, we rely on its existing Java model to PCM transformations to provide us with an updated PCM, on which we can apply PCM changes. Therefore, for the current state, we only consider PCM elements that are supported by CIPM. We define the transformations for Java code that complies with the coding conventions from the Java Language Specification [15]. As a consequence, our initial set of transformations represents one of numerous possibilities. They can be replaced, extended, and supplemented by other transformations, which are, for instance, specific for a certain project or technology.

Our initial set of transformations start with a component in the PCM. If it is newly created, we search for a Java module with the same name. If there is no module, we create one with the same name and add a correspondence between the component and module (cf., Figure 3a for pseudocode). For the lifetime of the correspondence, we keep the name of the component and module the same. Next, upon the creation of a PCM interface with a name, we check if it has a corresponding classifier element (i.e., class, interface, enumeration, or annotation) in the Java model by searching for classifiers with the same name. If there are multiple ones in different packages, we ask the software architect for the selection of a classifier. If we find no potential classifier, we create a new Java interface in the model. As the PCM does not directly support packages, we also ask the software architect for a package name and set it accordingly in the Java model. Then, we add a correspondence between the PCM interface and Java classifier. As long as this correspondence exists, we also keep the names of both elements the same. Analogously to the previous two PCM elements, the transformations handle service signatures in a PCM interface, parameters in the service signatures, their parameter types, and return types of service signatures. Regarding the deletion of a PCM element, the transformation is not trivial since it can affect multiple Java elements, of which it is hard to determine if they can be deleted, too. Even when the software architect is asked, the decision is not easy. Thus, we leave transformations for deletions for future work.

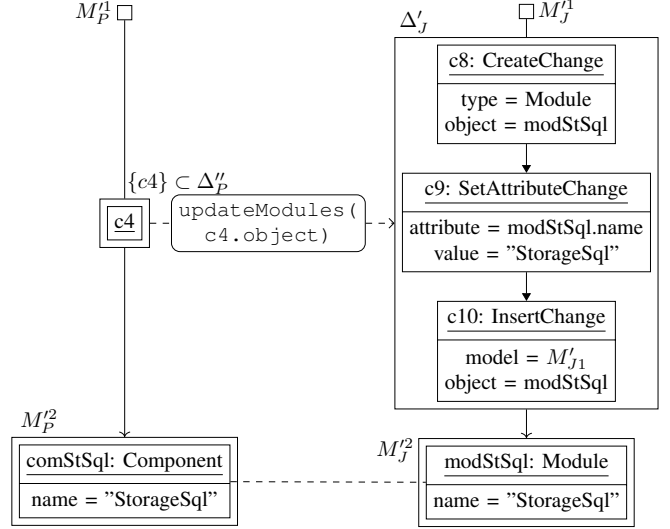
Figure 3b illustrates how the consistency preservation for a change in our running example could be processed. Into the empty model M_P^1 , the new storage component is inserted by change $c4$ (from Figure 2), leading to model M_P^2 . $c4$ triggers the transformation `updateModules`, outlined in Figure 3a,

```

1 function updateModules (
    newComp: Component) {
2   module ← createModule()
3   module.name ← newComp.name
4   addToJavaModel (module)
5   addCorrespondence (module, newComp) }

```

(a) Pseudocode for the consistency preservation of a newly created component.



(b) Example for applying the transformation `updateModules` from Figure 3a on $c4$ from Figure 2.

```

module teammates.storage.sql {}

```

(c) Example for generated code for model M_J^2 .

Fig. 3: Example for the consistency preservation.

to create a new module in the Java model corresponding to the new component. The resulting changes $c8$, $c9$, and $c10$ in the set Δ_J' for the model M_J^1 transform it into the model M_J^2 with the newly created module.

D. Manual User Interactions

Throughout our approach, we consider the software architect and developer as responsible people for decisions related to their respective model. When we cannot fully automatically determine how to maintain the models' consistency during the APM change propagation, we integrate both roles into the APM change propagation. Depending on the concrete scenario, we show a prompt to the software architect and perform the APM change propagation semi-automatically. Otherwise, we leave a task for the software developer to adapt the source code to APM changes after the change propagation. Communication between both roles is part of future work.

In our context, we define a prompt as a form of manual user interaction, where we pause the APM change propagation and ask the software architect for input. This means that the information the APM change propagation requires from the software architect determines the automaticity of the APM

change propagation. Therefore, it is important to examine what information we may need during the APM change propagation. Here, we distinguish between four types of information: (1) Specifications about the source code with corresponding APM elements, which are not provided in the APM; (2) Specifications regarding tagged correspondences between the APM and source code model; (3) Choices between multiple, applicable actions in response to an APM change; (4) Description for a concrete developer task associated with an APM change.

Given this classification, prompts can be automated during the APM change propagation if we can automatically acquire or derive the needed information. To this end, we introduce the *User Interaction Manager (UIM)*, which attempts to automate incoming prompts. To increase the effectiveness of UIM, we provide it with information on the APM and source code model throughout the APM change propagation, such as answers to previous prompts. We equip the UIM with different strategies to address certain prompts. These strategies can include heuristics or LLMs.

Since all APM changes are present prior to the APM change propagation and contain information on the APM, the integration of the UIM into the APM change preprocessing can improve the UIM. For example, we assume two classifiers with the same name in the Java model and each classifier contains a field, which have different names. If we now create a data type in the PCM with the same name as the two classifiers, the APM change propagation shows a prompt to manually select between the two classifiers. However, if an inner declaration is additionally introduced to the PCM data type with the same name as one of the fields in the two classifiers, the previous prompt is not necessary as the classifier can be unambiguously found.

E. Updating Source Code in a Version Control Repository

In the last step, we change the actual source code in the version control repository based on the updated source code model. From this model, we can generate source code via a model-to-text transformation. Here, different granularity levels are possible. For example, the whole source code can be generated from the model. Nevertheless, this is not desired in most cases. Instead, only source code files, whose corresponding model elements have changed, should be generated anew, or the exact code locations, where the source code model changed, are updated. After the code generation, the source code can be committed to a new branch in the version control repository. This allows software developers to review the changed code parts and implement missing parts. When they have finished their work, it can be merged to the main line, from which CIPM, for instance, picks up new commits and propagate them back to the APM. An example for generated code is shown in Figure 3c.

F. Implementation

We implemented parts of the aforementioned approach prototypically as an extension to CIPM. Specifically, for the preprocessing step, we consider a trivial change pattern and

change sequence transformation. We transform the change sequence to delay the introduction of new elements into the APM so that the APM stays as small as possible as long as possible.

For the transformations from the PCM to the Java model, we implemented the transformations described in subsection IV-C for PCM interfaces, service signatures, their return types, parameters, and parameter types. Although a code generation is provided by the extended JaMoPP [10], we did not include it yet in our implementation.

V. INITIAL EVALUATION

To evaluate our approach, we provide an initial experiment for our current prototypical implementation, outlined in subsection IV-F. The experiment builds upon a Goal Question Metric (GQM) plan [16] (subsection V-A) and experimental setup (subsection V-B), both also serving as a foundation for an extended future evaluation. In the following, beside the GQM plan and setup, we describe the first results (subsection V-C) and discuss them (subsection V-D).

A. GQM Plan

This section presents our GQM plan, which is a methodology for empirical evaluations [16]. We define our evaluation goals, corresponding questions, and metrics. Based on the measured metrics, we can answer the questions and decide if we achieved the goals.

Accuracy of APM change propagation (G1): As the source code model is updated, its accuracy is important to avoid follow-up errors and provide software developers valuable support. Since CIPM additionally requires an updated IM to calibrate the performance parameters of the APM, our CIPM extension also needs to propagate APM changes to the IM accurately. Although the APM to IM change propagation is already implemented in CIPM, which we reuse, we still evaluate its accuracy, as our CIPM extension acquires APM changes differently.

How accurate is the propagation of APM changes to the source code model? (G1-Q1) Here, we reuse a metric from a previous CIPM evaluation [7] and interpret a source code model as a set of model elements, for which we calculate the **Jaccard Similarity Coefficient (M1)** [17] between an updated model and reference model: $JC(A, B) := \frac{|A \cap B|}{|A \cup B|} \in [0, 1]$, where A and B are source code models. In this context, we match equal elements from both models based on their locations in the model structure, values, and types [7]. The equal elements form the intersection, while the remaining elements combined with the equal elements form the union.

How accurate is the propagation of APM changes to the IM? (G1-Q2) To evaluate the accuracy of the updated IM, we also reuse a metric from a previous CIPM evaluation [7]. Concretely, we use the **F1-Score (M2)** [18]: $F1 - Score(APM, IM) := \frac{2TP}{2TP + FN + FP} \in [0, 1]$. For each probe in the IM, reflecting a changed location for instrumentation, there should be a corresponding behavioral element in the PCM [7]. Therefore, we check these elements and their

correspondences. As a consequence, we define TP as the number of probes in the IM with a corresponding behavioral element, FN as the number of behavioral elements without a corresponding probe, and FP as the number of probes without a corresponding behavioral element.

How accurate is the propagation of APM changes to the APM? (G1-Q3) While we focus on the accuracy of the source code model and IM, we still want to ensure that the APM changes are accurately applied on the APM. Thus, we reuse the **Jaccard Similarity Coefficient (M1)** for an updated APM and reference model.

Automatability of APM change propagation (G2): To assess the adequacy of our approach for DevOps practices, where automaticity is crucial, we investigate the automaticity of our approach during the APM change propagation. As the APM change propagation to the IM is fully automatic, we focus on the change propagation to the source code model.

How automatable is the propagation of APM changes to source code models? (G2-Q1) As previously mentioned, the APM change propagation utilizes user interactions to clarify ambiguities. Since only such user interactions make the APM change propagation potentially not fully automatic, we count the number of user interaction occurrences and classify them whether they are fully automatic. Here, we consider a user interaction occurrence as fully automatic if it requires no manual input from a user. As a result, we define the metric **Automaticity Degree (M3)** for assessing the automaticity: $AD := \frac{|UI_{FA}|}{|UI_{ALL}|} \in [0, 1]$, where UI_{FA} is the set of all fully automatic user interaction occurrences and UI_{ALL} is the set of all user interaction occurrences during the APM change propagation. For completeness, we define $AD = 1$ if $UI_{FA} = UI_{ALL} = \emptyset$. We do not weight individual user interactions based on their required manual effort to keep M3 objective.

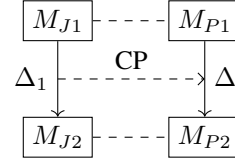
Performance of APM change propagation (G3): We intend our CIPM extension to help accelerate software development in DevOps environments. Therefore, it needs to propagate APM changes within a reasonable time frame so that our approach can be applied in practice.

How much time does it take to propagate APM changes to the source code model and to the IM? (G3-Q1) To answer this question, we use the **execution time of the APM change propagation in seconds (M4)** as metric.

B. Experimental Setup

Based on our GQM plan, we designed an initial experiment that builds upon TEAMMATES [12], which is one of the existing evaluation cases in CIPM [7]. Within CIPM's evaluation [7], five selected commits of TEAMMATES are input into the consistency preservation from source code to the APM. For the first commit¹, the complete source code of TEAMMATES is parsed and propagated to an empty PCM. For the remaining commits, the source code changes between two commits are

1. Execution of CIPM



2. Experiment

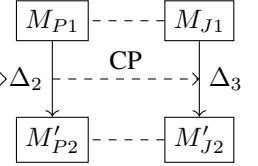


Fig. 4: The general two-phased flow of our experiment. We omitted the IM for clarity.

propagated to the existing PCM of the older commit. For our initial experiment, we set two limits: (1) We only employ the aforementioned first commit of TEAMMATES considered in CIPM, and (2) we partially disable the behavior reconstruction for the PCM.

The general two-phased flow of our experiment is as follows and shown in Figure 4: We start by executing CIPM (phase 1) with two commits C_1 and C_2 . For C_1 , there are already a source code model M_{J1} and a corresponding PCM M_{P1} and IM_1 . CIPM applies the changes Δ_1 between C_1 and C_2 to the source code model M_{J1} , leading to M_{J2} . At the same time, this preserves the consistency to the PCM and IM by generating changes Δ_2 and Δ_{IM2} , leading to M_{P2} and IM_2 , respectively. We store all changes, with which we automatically obtained APM changes for the actual experiment in the next phase 2. There, we use our implemented approach to apply Δ_2 on M_{P1} to simulate the development of the PCM. The application of Δ_2 results in an updated PCM M'_{P2} and the consistency preservation towards the source code model and IM with changes Δ_3 , updated source code model M'_{J2} , changes Δ'_{IM2} , and updated IM'_2 . During the experiment, we measure the execution time of the APM change propagation (M4 for G3-Q1) and compute the automaticity degree AD (M3 for G2-Q1). Afterward, we calculate $JC(M_{J2}, M'_{J2})$ for G1-Q1, $F1 - Score(M'_{P2}, IM'_2)$ for G1-Q2, and $JC(M_{P2}, M'_{P2})$ for G1-Q3. In addition, we determine the same metrics for the results of the first phase, according to CIPM's evaluation [7], to validate the first phase's results.

To avoid influences by potential user interactions, we modify the UIM to fully automate all user interactions the original UIM cannot automate, described by the set UI_M of all manual user interactions so that UI_M consists of all user interaction occurrences without the fully automatable ones: $UI_M = UI_{ALL} \setminus UI_{FA}$. As a result, manual user interactions have a lower effect on M4. To still measure M3 accurately and investigate the potential of automaticity, we consider UI_M as manual user interaction occurrences.

C. Results

We executed our initial experiment on a laptop with an Intel(R) Core(TM) i7-3520M CPU @ 2.90 GHz, 16 GB RAM, Windows 10 Pro, and Java 11. In the experimental run $Java \rightarrow PCM$, CIPM computed 570,622 changes for the source code model, 14,879 changes for the PCM, and 4,665 changes for the IM. The whole run took ~ 657.57 seconds with a

¹Commit 648425: <https://github.com/TEAMMATES/teammates/commit/648425746bb9434051647c8266dfab50a8f2d6a3>

resulting $JC(M_{J_2}, M_{J_2}^R) = 1$ ($M_{J_2}^R$ is a reference model for M_{J_2}), $F1 - Score(M_{P_2}, IM_2) = 1$, and $AD = 1$. Afterward, our implemented approach propagated all 14,879 PCM changes within ~ 19.61 seconds, resulting in 4,983 Java model changes and 4,665 IM changes. Our execution recorded $|UI_{ALL}| = 196$ issued user interactions, of which four user interactions were addressed by the original UIM and the rest were addressed by the modified UIM. This leads to $|UI_M| = 192$, $|UI_{FA}| = 4$, and $AD \approx 0.0204$. In terms of accuracy, our implemented approach achieved $JC(M_{P_2}, M_{P_2}') \approx 0.9994$, $F1 - Score(M_{P_2}', IM_2') = 1$, and $JC(M_{J_2}, M_{J_2}') \approx 0.0011$.

D. Discussion

Our experimental results highlight that the Java to PCM propagation included significantly more changes than the PCM to Java propagation. Congruently, the Java to PCM propagation took longer than the PCM to Java propagation. We expected this effect since source code models usually contain many model elements, which represent low-level code details that are not relevant for APMs. In accordance with CIPM's evaluation results [7], the Java to PCM propagation was fully accurate and automatic.

We expected that the PCM to Java propagation has a lower accuracy for the source code model, compared to the Java to PCM propagation. By looking into the generated Java model, we observed multiple causes for the accuracy results: (1) Source code usually consists of mainly method bodies and statements, for which the extended JaMoPP uses numerous elements to represent them. Therefore, by neither propagating changes of behavioral elements to the source code model nor including Java elements for method bodies and statements in the transformations, our experiment can lower the source code model accuracy severely. (2) The creation of a Java model is a challenging process, since the parser needs to be robust and account for dependencies in the source code, including external libraries [9], [10]. As our approach is designed to propagate APM changes and not only for creating source code models from scratch, it is not directly suited to address the creation of Java models from scratch and the challenge of modeling the various dependencies in the source code model. (3) CIPM uses TEAMMATES' first commit to build a source code model from scratch. Consequentially, each of the Java to PCM and the PCM to Java propagations started with an empty Java model. While the Java to PCM propagation can successfully build a complete Java model and the PCM from scratch as expected, the PCM to Java propagation only propagates the generated PCM changes to an empty Java model, which excludes many aspects of source code, such as dependencies (2) or method bodies (1).

In terms of automaticity, we knew that there are no user interactions for the Java to PCM propagation [7], with which our experiment results align and which we used as a baseline. For the PCM to Java propagation, we expected little to no automaticity before the experiment execution, since $PCM \rightarrow Java$ starts with empty models. As a result, this leaves a large

design space for the PCM to Java propagation, leading to manual user interactions. This is further complicated by the source code model, which contains no model elements for external libraries.

VI. RELATED WORK

Bidirectional transformation systems, such as TGGs or QVT-R, enable the definition of bidirectional model transformations to maintain the consistency between two models [19]. In contrast, our and the CIPM approach employ unidirectional transformations. Due to the different abstraction levels of APMs and source code and depending on their concrete mappings, certain transformations cannot be expressed as bidirectional transformations or are not invertible. For example, the behavior reconstruction in the APM from source code [7], which abstracts source code statements to higher-level behavior elements, cannot be reversed as information is lost.

Langhammer [20] also tackles the propagation of APM changes to source code and vice versa. They introduced exemplary mappings between APM and source code for different technologies, consisting of bidirectional transformations. Their proposed process targets the consistency preservation directly during software development in an IDE. While our approach is similar to Langhammer's one, there are also conceptual differences: (1) They do not consider composite PCM changes during the APM change propagation. (2) They immediately propagate PCM changes to Java without preprocessing them. (3) Their approach is directly integrated into model editors, from which they receive and immediately propagate PCM changes. (4) We consider PCMs generated by CIPM, while Langhammer's approach supports integrating artifacts outside the approach. (5) They introduce more specific PCM to Java transformations compared to our initial set of transformations. For instance, they assign a Java classifier to each individual PCM component. As a consequence of these differences, Langhammer's and our approach suit different purposes: Langhammer's approach is designed for immediate IDE-integrated APM-code-synchronization, and our approach deals with round-trip engineering of APMs in DevOps.

Cavalcante et al. [21] developed an approach to generate initial source code in the Go programming language from the π -ADL in a batch process. In contrast, our approach incrementally updates Java source code based on changes in an APM.

Di Rocco et al. [22] propose an approach to handle breaking and unresolvable changes of metamodels when co-evolving their models by involving developers. This work can serve as a foundation for future work regarding the manual user interactions.

As change preprocessing is a form of semantic lifting, semantic lifting by Kehrer et al. [14] is relevant to our work. It aims to derive high-level changes from low-level changes. In this context, we can consider high-level changes as composite changes and low-level changes as atomic changes. In their

work, Kehrer et al. [14] use a state-based model comparison to compute model changes, which are then processed based on pattern-based rules to compute change sets. Since multiple rules could apply to certain change sequences and introduce ambiguities, they consider multiple potential change sets and post-process them for disambiguation. Metamodel complexity for models and model changes can complicate the pattern recognition throughout change sequences. Thus, the disambiguation process is not trivial to realize in general. In our case, we can embed semantic lifting into the APM change preprocessing in future work. High-level / composite changes can also support software architects and developers in understanding change sequences since they do then not only consist of low-level / atomic changes.

VII. CONCLUSION

APMs offer a high-level view on source code and enable performance predictions to explore design alternatives [3]. To this end, APMs need to represent the corresponding code accurately, which can be achieved with the continuous round-trip engineering between APM and source code. Thus, in this paper, we presented our current work in progress towards such a process. We build upon the CIPM approach [7], which continuously updates an APM to source code changes in a DevOps environment. Our approach extends CIPM with the propagation of APM changes to source code. It obtains APM changes, preprocesses them, and propagates them via incremental, rule-based model transformations (semi-)automatically to a source code model, from which source code can be generated. We provided an initial evaluation for our current prototypical implementation.

As our next steps, we plan to extend the implementation in the various steps of our approach. This includes advanced change preprocessing and addressing potential ambiguities after preprocessing. Furthermore, we aim to include more transformations for more types of APM changes and to improve the UIM to better automate potential manual user interactions. Based on the extended implementation, we want to evaluate our approach on further commits and projects.

In future work, we want to investigate how well LLMs can support the consistency preservation between the APM and source code. We suggest two ways: either let an LLM generate the transformations or let an LLM directly adapt the source code based on the APM changes. We hypothesize that the availability of the APM changes can positively impact the LLMs.

ACKNOWLEDGMENT

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - SFB 1608 - 501798263 and supported by the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF).

REFERENCES

- [1] C. Verbruggen and M. Snoeck, "Practitioners' experiences with model-driven engineering: a meta-review," *Software and Systems Modeling*, vol. 22, no. 1, pp. 111–129, Feb 2023. [Online]. Available: <https://doi.org/10.1007/s10270-022-01020-1>
- [2] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, p. 40–52, Oct. 1992. [Online]. Available: <https://doi.org/10.1145/141874.141884>
- [3] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Kozirolek, H. Kozirolek, M. Kramer, and K. Krogmann, Eds., *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016.
- [4] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, "A survey of devops concepts and challenges," *ACM Comput. Surv.*, vol. 52, no. 6, Nov. 2019. [Online]. Available: <https://doi.org/10.1145/3359981>
- [5] E. Anthony, A. Berntsson, T. Santilli, and R. Wohlrab, "We're drifting apart: Architectural drift from the developers' perspective," in *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, 2024, pp. 101–111. [Online]. Available: <https://doi.org/10.1109/ICSA59870.2024.00018>
- [6] T. Hettel, M. Lawley, and K. Raymond, "Model synchronisation: Definitions for round-trip engineering," in *Theory and Practice of Model Transformations*, A. Vallecillo, J. Gray, and A. Pierantonio, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 31–45. [Online]. Available: https://doi.org/10.1007/978-3-540-69927-9_3
- [7] M. Mazkatli, D. Monschein, M. Armbruster, R. Heinrich, and A. Kozirolek, "Continuous integration of architectural performance models with parametric dependencies—the cipm approach," *Automated Software Engineering*, vol. 32, no. 2, pp. 1–79, 2025. [Online]. Available: <https://doi.org/10.1007/s10515-025-00521-9>
- [8] F. Gorsler, F. Brosig, and S. Kounev, "Performance queries for architecture-level performance models," in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 99–110. [Online]. Available: <https://doi.org/10.1145/2568088.2568100>
- [9] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, "Closing the gap between modelling and java," in *Software Language Engineering*, M. van den Brand, D. Gašević, and J. Gray, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 374–383. [Online]. Available: https://doi.org/10.1007/978-3-642-12107-4_25
- [10] M. Armbruster, "Parsing and printing java 7-15 by extending an existing metamodel," Tech. Rep., 2022. [Online]. Available: <https://doi.org/10.5445/IR/1000149186>
- [11] H. Klare, M. E. Kramer, M. Langhammer, D. Werle, E. Burger, and R. Reussner, "Enabling consistency in view-based system development — the vitruvius approach," *Journal of Systems and Software*, vol. 171, p. 110815, 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121220302144>
- [12] "Teammates developer web site," accessed: 2026-03-23. [Online]. Available: <https://github.com/TEAMMATES/teammates/tree/db42a206355e64816ce5759de3a4f43197eb7825>
- [13] C. Brun and A. Pierantonio, "Model differences in the eclipse modeling framework," *UPGRADE, The European Journal for the Informatics Professional*, vol. 9, no. 2, pp. 29–34, 2008.
- [14] T. Kehrer, U. Kelter, and G. Taentzer, "A rule-based approach to the semantic lifting of model differences in the context of model versioning," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, Nov 2011, pp. 163–172. [Online]. Available: <https://doi.org/10.1109/ASE.2011.6100050>
- [15] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, "The java language specification, java se 15 edition." [Online]. Available: <https://docs.oracle.com/javase/specs/jls/se15/jls15.pdf>
- [16] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, pp. 728–738, 11 1984. [Online]. Available: <https://doi.org/10.1109/TSE.1984.5010301>
- [17] A. Grygorian and I. E. Iacob, "A concise proof of the triangle inequality for the jaccard distance," *The College Mathematics Journal*, vol. 49, no. 5, pp. 363–365, 2018. [Online]. Available: <https://doi.org/10.1080/07468342.2018.1526020>
- [18] C. Goutte and E. Gaussier, "A probabilistic interpretation of precision, recall and f-score, with implication for evaluation," in *Advances in Information Retrieval*, D. E. Losada and J. M. Fernández-Luna, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 345–359. [Online]. Available: https://doi.org/10.1007/978-3-540-31865-1_25
- [19] S. Hidaka, M. Tisi, J. Cabot, and Z. Hu, "Feature-based classification of bidirectional transformation approaches," *Software & Systems*

- Modeling*, vol. 15, no. 3, pp. 907–928, Jul 2016. [Online]. Available: <https://doi.org/10.1007/s10270-014-0450-0>
- [20] M. Langhammer, *Automated Coevolution of Source Code and Software Architecture Models*. Karlsruhe: KIT Scientific Publishing, 08 2019. [Online]. Available: <https://doi.org/10.5445/KSP/1000081447>
- [21] E. Cavalcante, F. Oquendo, and T. Batista, “Architecture-based code generation: From π -adl architecture descriptions to implementations in the go language,” in *Software Architecture*, P. Avgeriou and U. Zdun, Eds. Cham: Springer International Publishing, 2014, pp. 130–145. [Online]. Available: https://doi.org/10.1007/978-3-319-09970-5_13
- [22] J. Di Rocco, D. Di Ruscio, A. Pierantonio, and L. Iovino, “Supporting users to manage breaking and unresolvable changes in coupled evolution,” in *Proceedings of the Workshop on Domain-Specific Modeling*, ser. DSM 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 47–54. [Online]. Available: <https://doi.org/10.1145/2846696.2846703>