

CoCoPath: Concolic Exploration of Consistency-Preserving Paths

Bowen Jiang*, Miriam Boss*, Thomas Weber*, Weixing Zhang*, Mattias Ulbrich*, and Anne Kozirolek*

*Karlsruhe Institute of Technology, Germany

1 ABSTRACT

Maintaining consistency between related models is achieved through Consistency Preservation Rules (CPRs) that propagate changes from a source model to a target model. However, in many cases, additional input from the maintainer of a target model, based on their domain knowledge, is required to integrate changes into target models. Therefore, the change cannot be propagated automatically and introduces temporary inconsistency. To manage the temporary inconsistency, we propose CoCoPath, a concolic execution framework for systematically exploring execution paths in consistency-preserving model transformations. We combine dynamic taint tracking, concolic execution, and model transformation frameworks to derive potential target models based on the source model, the consistency preservation rules, and optional domain constraints to further restrict the search space. By comparing these alternative target models, users can explore the consequences of different additional inputs and resolve temporary inconsistency in an informed manner. We validate our contribution by applying our approach to an industry case study, showcasing its capabilities for one and multiple awaited user inputs.

KEYWORDS Consistency Preservation, Model-Driven Engineering, Model Transformation, Taint Tracking, Concolic Execution, Path Exploration, Dynamic Analysis.

1. Introduction

When a system’s architectural model declares “Task A has priority five”, but the implementation code shows “Task A has priority three”, we say these two models are inconsistent. In Model-Driven Engineering (MDE), consistency refers to the state in which two or more overlapping elements in different models of the same system have a satisfactory joint description (Lucas et al. 2009; Zhang et al. 2025). Maintaining such cross-model consistency is a fundamental challenge in view-based system development (Bruneliere et al. 2019; Cicchetti et al. 2019).

One approach to consistency problems is to use a Single Underlying Model (SUM)—a complete system description that

contains all information about the system with no redundancy and no implicit dependencies (Atkinson et al. 2008). In such a SUM, all views are generated dynamically through projection, thus naturally guaranteeing consistency. However, defining such a SUM for complex systems faces two fundamental difficulties: first, defining a redundancy-free monolithic metamodel is itself extremely challenging (Meier et al. 2019); second, this approach cannot reuse existing domain-specific metamodels and tool chains (Klare et al. 2021). To address this, the VITRUVIUS approach proposes the concept of Virtual Single Underlying Model (VSUM) (Klare et al. 2021). VSUMs adopt a modular structure, composed of multiple existing metamodels that may contain redundancies, coupled through explicitly defined Consistency Preservation Rules (CPR). While internally maintaining consistency through CPRs, VSUMs externally behave as contradiction-free system descriptions. However, in practice, many CPRs cannot be executed fully automatically—they require input or decisions from the maintainer of a target model based on their domain knowledge. Without the required user input, temporary inconsistency is introduced, posing a ques-

JOT reference format:

Bowen Jiang, Miriam Boss, Thomas Weber, Weixing Zhang, Mattias Ulbrich, and Anne Kozirolek. *CoCoPath: Concolic Exploration of Consistency-Preserving Paths*. Journal of Object Technology. Vol. 25, No. 3, 2026. Licensed under Attribution 4.0 International (CC BY 4.0) <http://dx.doi.org/10.5381/jot.2026.25.3.a13>

tion for developers: how to make informed decisions without knowing the consequences of alternative consistency-preserving paths?

Consider an example from automotive software development, in which engineers work with AMALTHEA models¹ for modeling multicore software and with Advanced Simulation and Control Engineering Tool (ASCET) (Lefarth et al. 1998; ASCET Repository 2024) for modeling embedded control systems. When an engineer adds a new task to an AMALTHEA architecture model, the corresponding CPR needs to create a task implementation in the ASCET model (Mazkatli et al. 2017). However, the abstract Task type in AMALTHEA must be mapped to one of four concrete task subtypes in ASCET, which requires the engineer to make a decision based on domain knowledge that cannot be inferred automatically. This decision is therefore modeled as a parameter of the corresponding CPR. In this situation, the question arises: what are the possible paths formed by different user , and how can engineers reason about them systematically? While we use AMALTHEA and ASCET as an example and case study, the problem of user-driven branching during consistency preservation arises in many multi-model engineering scenarios where mappings are underspecified or context-dependent.

To address these challenges, we propose CoCoPath, which, to the best of our knowledge, is the first concolic execution framework for systematically exploring execution paths in consistency-preserving model transformations. The contribution of this paper is a novel combination of dynamic taint tracking, concolic execution, and model transformation frameworks that enables systematic exploration of the decision space. Specifically, CoCoPath treats awaited user inputs as symbolic variables and marks them with symbolic tags before execution, while automatically propagating symbolic information as the transformation logic executes. Path constraints are collected at control flow decision points, then negated and solved using an SMT solver (De Moura & Bjørner 2008) to systematically explore different decision branches. CoCoPath employs a hybrid constraint collection framework that combines two complementary mechanisms for constructing path constraints, optionally refined by domain constraints. For each explored path, CoCoPath generates the corresponding target model, symbolic input combinations, and path constraint sets. To provide a precise foundation for this exploration, we further extend the formalization of CPR to integrate user inputs, which have not been covered in the formalizations we build upon and serve as the conceptual basis for CoCoPath. We implement a CoCoPath prototype within the VITRUVIUS framework, using Galette (Hough & Bell 2025) for dynamic taint tracking and GreenSolver (Visser et al. 2012) as the constraint solver, and validated the approach’s effectiveness through two case studies. By automatically exploring all feasible paths, our method generates concrete result models for each option, allowing engineers to directly compare the model states resulting from different choices. By quantitatively analyzing the model changes caused by each option (such as the number of added, deleted, and modified elements), engineers can identify high-impact decision points and low-impact

¹ <https://eclipse.dev/app4mc/help/latest/index.html#section3>

choices. Furthermore, complete path exploration can reveal the feasibility of different decision combinations, helping engineers understand the structure of the decision space. This enables engineers to reason about decision consequences before committing to a repair, rather than reacting to unintended model changes after the fact.

To systematically evaluate our approach, we address the following research questions (RQs):

RQ1: Can CoCoPath systematically explore all feasible execution paths induced by user decisions, and to what extent are the resulting outcomes analyzable? Existing consistency preservation approaches do not provide systematic support for exploring all feasible execution paths induced by user inputs, leaving engineers without a comprehensive overview of possible transformation outcomes.

RQ2: What is the runtime performance and scalability of multi-variable path exploration? Since concolic execution introduces additional runtime overhead through symbolic propagation and constraint solving, we assess whether the approach scales to multi-variable decision spaces.

RQ3: How does CoCoPath compare to existing symbolic execution tools with respect to their applicability to consistency-preserving model transformations in VITRUVIUS frameworks? Existing symbolic execution tools such as Java PathFinder, CATG, and JDart are designed for general-purpose Java programs and have not been evaluated for their compatibility within MDE frameworks such as VITRUVIUS.

RQ4: What is the memory overhead of CoCoPath? Dynamic taint tracking requires maintaining shadow state alongside concrete values, and constraint solving introduces additional data structures. The resulting memory overhead in the context of model transformations has not been previously quantified.

The remainder of this paper is structured as follows: Section 2 provides background. Section 3 discusses related work. Section 4 presents the CoCoPath approach. Section 5 describes the prototypical implementation. Section 6 evaluates CoCoPath through the AMALTHEA-ASCET case study and through a BrakeSystem-ControlSystem case study and discusses threats to validity. Section 7 discusses findings and limitations. Section 8 concludes the paper.

2. Background

We introduce the background concepts underlying CoCoPath: CPR formalization, dynamic taint tracking, and concolic execution.

2.1. Formalization of CPRs

In the following, we base our formalization on Pascual et al. (2024) as their approach is more lightweight and sufficient for our purpose. For a formalization that provides a more in-depth approach to consistency, we refer to Klare et al. (2021).

VSUM A metamodel M defines which syntactical constructs are well-formed models. Analogous to a formal grammar G producing a formal language $L(G)$ of words, M produces a set $L(M)$ of models. For the sake of readability, we identify $L(M)$ with M in the following. The set of models within a VSUM

1 metamodel is structured as a Cartesian product $\mathbf{M} = M_1 \times$
2 $\dots \times M_n$, accompanied by a relation $CR \subseteq \mathbf{M}$ determining
3 which tuples of models within \mathbf{M} are considered consistent.
4 An instance of a VSUM is defined as \mathbf{m} where $\mathbf{m} \in \mathbf{M}$ and
5 $\mathbf{m} = (m_1, \dots, m_n)$, $\{m_i \in M_i \mid 1 < i < n\}$. For simplicity,
6 we consider a VSUM metamodel $\mathbf{M} = M_1 \times M_2$, consisting
7 of two individual metamodels M_1 and M_2 . The extension to an
8 arbitrary number of metamodels is straightforward.

9 **CPR** A CPR c reacting to changes in M_1 and M_2 is a function
10 $c : M_1 \times M_1 \times M_2 \rightarrow M_2$. Given a transition from m_1 to
11 m'_1 in M_1 and a model $m_2 \in M_2$, the output is the updated
12 model $m'_2 \in M_2$, i.e., $c(m_1, m'_1, m_2) = m'_2$. This function thus
13 proposes a transition from m_2 to m'_2 to reestablish consistency.
14 A CPR c is called *correct wrt.* CR if $(m_1, m_2) \in CR$ implies
15 $(m'_1, c(m_1, m'_1, m_2)) \in CR$ for all $m_1, m'_1 \in M_1$ and $m_2 \in$
16 M_2 . We call the temporarily inconsistent VSUM $\mathbf{m}_{source} =$
17 (m'_1, m_2) the source VSUM, while the VSUM with restored
18 consistency $\mathbf{m}_{target} = (m'_1, m'_2)$ is called the target VSUM.

19 **Path** In this context, an (execution-)path is an instance of an
20 applied correct CPR per input combination, i.e., with the CPR
21 definition there is exactly one execution path per input combi-
22 nation. With the integration of user input, an input combination
23 may have multiple paths. From a more technical standpoint, an
24 (execution-)path is a concrete control flow path through a trans-
25 formation execution, determined by the outcomes of conditional
26 statements that depend on user input. Additionally, path con-
27 straints can be used to impose further restrictions on behavior
28 of a CPR on branching points. We call each conditional control-
29 flow construct whose outcome depends on a CPR parameter a
30 decision point. An execution path is then determined by the
31 sequence of outcomes at all decision points encountered during
32 a CPR execution.

33 2.2. Symbolic Execution and Concolic Execution

34 To systematically collect all possible paths, we employ concolic
35 execution, which is also called dynamic symbolic execution.
36 Concolic execution is the combination of symbolic and con-
37 colic execution, which been developed to create test inputs to
38 achieve branch coverage in code (Godefroid et al. 2005): With
39 the help of concrete inputs, symbolic constraints are derived;
40 while symbolic constraints are used to reduce the search space.
41 This helps mitigate limitations imposed by (SMT) solvers to be
42 able to explore the entire search space automatically. Recent
43 approaches that are developed for fuzzing, i.e., creating test
44 inputs to increase code coverage, include JDart (Luckow et al.
45 2016) and CONFETTI (Kukucka et al. 2022). However, these
46 approaches only support Java 8. We have repurposed a part
47 of CONFETTI, namely Knarr, for our approach and integrated
48 parts of it into Galette, which supports modern Java.

49 2.3. Dynamic Taint Tracking

50 Taint tracking originates from the security domain, where po-
51 tentially malicious inputs are “tainted”, i.e., tagged, to track
52 where they are used, and whether any critical function might
53 be affected. *Dynamic* taint tracking denotes that tags are prop-
54 agated during program execution. It is also applicable for the

55 general purpose of tracking flows through a program. For our
56 prototype, we use the Reactions language of VITRUVIUS, which
57 is compiled into Java bytecode; therefore, we use taint tracking
58 for the Java Virtual Machine (JVM). There are multiple taint
59 trackers available, e.g., Phosphor (Bell & Kaiser 2014), Mirror-
60 Taint (Ouyang et al. 2023), or Galette (Hough & Bell 2025). We
61 use Galette because it is the most compatible with modern JVM
62 versions, which are used in VITRUVIUS.

63 3. Related Work

64 **Consistency Preservation Tools** Several tools have been
65 developed to manage consistency between models, e.g., De-
66 signSpace (Demuth, Riedl-Ehrenleitner, et al. 2015), Open-
67 Flexo (Golra et al. 2016), MetaEdit+ (Kelly et al. 1996; Kelly &
68 Tolvanen 2021), Comprehensive Systems (Stünkel et al. 2021),
69 or openCAESAR (Elaasar et al. 2023). While they share com-
70 mon goals with VITRUVIUS in general and also with CoCoPath
71 and its goal of developer guidance, they differ in the way they
72 treat user input and in parts the explicit definition of consis-
73 tency rules. CoCoPath exploits both for concolic exploration of
74 consistency-preserving paths under user-driven changes, provid-
75 ing guidance to developers in their task to preserve consistency
76 in a multi-model environment.

77 **Model Transformation Approaches** For the broader field of
78 model transformations, several approaches deal with related
79 questions. Eramo et al. (2015) generate multiple consistent
80 models from an underspecified relational transformation in JTL
81 and represent them compactly using uncertainty models. In
82 contrast, CoCoPath analyzes the execution paths of procedural
83 consistency-preserving repair programs with symbolic inputs
84 to systematically explore possible repair outcomes. Egyed et al.
85 (2011) highlight the central role of human guidance in model
86 transformation, arguing that change propagation across hetero-
87 geneous models often cannot be fully automated due to missing
88 or ambiguous information in their vision of a smart transforma-
89 tion assistant. Kretschmer et al. (2021) systematically explore
90 sequences of repairs to models that have been changed by devel-
91 opers, similarly to CoCoPath. In contrast to their approach, we
92 exploit the, in our approach defined and used, consistency rules
93 for computing such sequences. Constraint-driven Modeling
94 (CDM) (Demuth, Lopez-Herrejon, & Egyed 2015) expresses
95 the implications of changes to one model through constraints
96 on another model. Laghouaouta & Laforcade (2020) investigate
97 uncertainty within model transformations by introducing partial
98 patterns, which allow a transformation rule to encode multi-
99 ple possible matching and production scenarios. Famelis et al.
100 (2013) employ *May Models* to explicitly model uncertainty in
101 models and lift graph-rewrite transformations to operate cor-
102 rectly on those models. Outside model transformations, efficient
103 exploration has also been studied in directed controller synthe-
104 sis (Ubukata et al. 2025).

105 **Symbolic Execution for Java** **Java Pathfinder** (Pășăreanu
106 & Rungra 2010) is a symbolic execution engine for Java byte-
107 code that systematically explores program paths by combining
108 concrete and symbolic execution to detect errors and verify

properties. Due to different requirements for the Java runtime, i.e., Java 8 for Pathfinder and 17 for VITRUVIUS, they are incompatible **TesMa and CATG** (Tanno et al. 2015) are automated test generation tools designed for enterprise application models, leveraging concolic execution and constraint solving to systematically explore execution paths and generate high-coverage test cases. TesMa and CATG focus on test generation for enterprise application models and not consistency management across heterogeneous models. **JDart** (Luckow et al. 2016) is a concolic execution framework for Java, developed by CMU and NASA. It combines concrete and symbolic execution, uses Java PathFinder, and supports multiple constraint solvers, enabling advanced features, e.g., method summarization. As it builds on Java Pathfinder, it is incompatible with VITRUVIUS due to different requirements for the Java runtime.

4. CoCoPath Approach

This section presents the methodology underlying CoCoPath. We first give an overview of the approach in Section 4.1. We then formalize parameterized consistency-preserving rules and the notion of execution paths induced by user inputs in Section 4.2. We explain how user decisions are represented as symbolic inputs and how symbolic information is propagated during concrete execution in Section 4.4. Next, we introduce two complementary mechanisms for constructing path constraints and discuss their respective trade-offs in Section 4.5. Finally, we describe the iterative concolic exploration loop that systematically explores all feasible execution paths in Section 4.6.

4.1. Overview

As established in Section 1, user inputs required during CPR execution may trigger different control-flow paths, leading to substantially different target models. Without systematic support, engineers must reason about these consequences manually, without knowing which alternatives are feasible or how they differ in impact.

Throughout this paper, we use the term *user input* to denote a value supplied by a human maintainer during transformation execution to resolve underspecified behavior in a CPR. In the formalization, such inputs correspond to parameters $t \in T$ of a parameterized CPR, explained in Section 4.2. During execution, user inputs are represented as symbolic inputs whose propagation and constraints are handled by CoCoPath.

CoCoPath enables systematic path exploration for CPRs that requires user inputs. Our key design principle is to *separate symbolic propagation from constraint construction*: (1) symbolic information is propagated uniformly during concrete execution, while (2) path constraints are constructed by dedicated mechanisms at decision points. This allows CoCoPath to adapt to different transformation stacks: the same symbolic propagation layer can be paired with different constraint construction mechanisms depending on runtime compatibility requirements.

Figure 1 illustrates the overall concolic exploration loop employed by CoCoPath. Starting from an initial temporarily inconsistent model state, the missing user input is selected as a symbolic variable (step 1) with default initial values. CoCoPath

then marks these inputs with symbolic tags (step 2) and executes the CPRs concretely (step 3) while propagating symbolic information (step 4). During execution, the taken control-flow path is observed and path constraints are constructed (step 5). After execution, our approach checks for unexplored paths (step 6) and negates selected constraints (step 7). If a satisfying assignment exists, the CPRs are re-executed under the new inputs to explore a different path (step 8). This process is repeated until no further feasible paths can be found.

4.2. Formalization of Parameterized CPRs

In the following, we extend the formalization introduced in Subsection 2.1 to integrate user inputs, so that we can use it for the CoCoPath approach.

Parameterized CPRs A CPR c_p may also be *parameterized*, in which case it incorporates an extra parameter from a parameter space T such that $c_p : M_1 \times M_1 \times M_2 \rightarrow (T \rightarrow M_2)$. Consequently, the application $c_p(m_1, m'_1, m_2)$ of a parameterized CPR c_p to models from the respective metamodels does not yield a single model m'_2 like in the unparameterized case, but is a function from the parameter type T to the models in M_2 , producing a single repaired model for each parameter selection.

A parameterized CPR is correct wrt. CR if $(m_1, m_2) \in CR$ implies $(m'_1, c(m_1, m'_1, m_2)(t)) \in CR$ and $m_2 \in M_2$ and additionally for all $t \in T$. As an example, a change in m_1 may trigger the introduction of a new element in m_2 . The name of that new element may then be the CPR-parameter and T be the space of admissible names.

The parameter space often depends on the input models, which can be expressed through dependently typed functions: $c_{dp} : (m_1 : M_1) \times M_1 \times (m_2 : M_2) \rightarrow T(m_1, m_2) \rightarrow M_2$. Here the parameter space $T(m_1, m_2)$ now depends on the input models m_1 and m_2 .

Path Exploration With the formal setting in place, we apply the formal framework to explore the path space in CoCoPath. CPRs function as model transformations implemented within a modeling framework. As such, they are programs in some programming language with some form of control flow. While the subsequent sections will focus on control structures within programs in the modeling framework, we shall maintain a higher level of abstraction for the moment. Whatever the means is to implement a CPR, thanks to the control flow mechanics of the implementation language, it can be represented as a case distinction enumerating all paths, as follows:

$$c_p(m_1, m'_1, m_2)(t) = \begin{cases} c_p^{(1)}(m_1, m'_1, m_2)(t) & \text{if } \varphi^{(1)}(m_1, m'_1, m_2, t) \\ c_p^{(2)}(m_1, m'_1, m_2)(t) & \text{if } \varphi^{(2)}(m_1, m'_1, m_2, t) \\ \dots & \\ c_p^{(k)}(m_1, m'_1, m_2)(t) & \text{if } \varphi^{(k)}(m_1, m'_1, m_2, t) \end{cases}$$

where k denotes the number of paths through the program and $\varphi^{(i)}$ (with $1 \leq i \leq k$) is the path condition for each path through the CPR implementation. The conditions $\varphi^{(i)}$ are mutually exclusive if they describe different paths of a deterministic

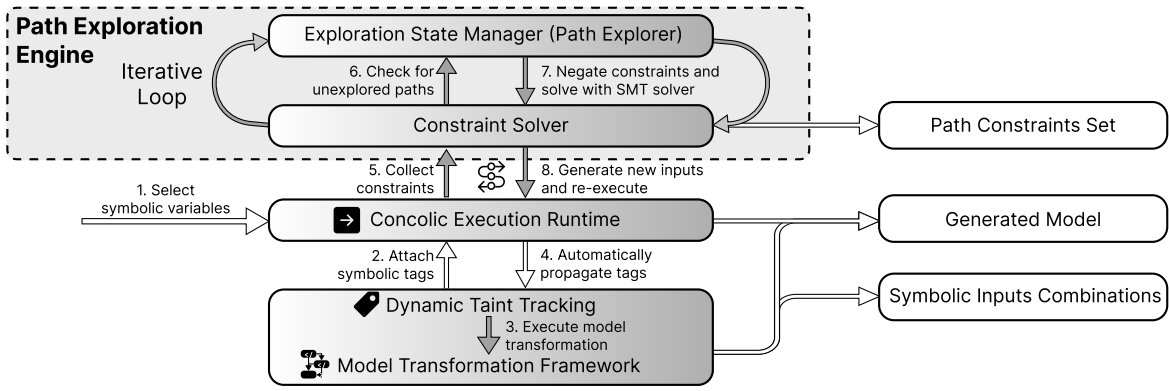


Figure 1 Overview of the CoCoPath approach

1 program. The function $c_p^{(i)}$ represents the program's effect for
 2 the i -th path through c_p .

3 The aim of the CoCoPath approach is the following:

4 For given $m_1, m'_1 \in M_1$ and $m_2 \in M_2$, find a set
 5 of parameters $T_0 \subseteq T$ such that $\{i \mid 1 \leq i \leq$
 6 $k, c_p^{(i)}(m_1, m'_1, m_2)(t), t \in T_0\}$ has maximum cardi-
 7 nality.

8 The resulting set of repair alternatives $\{c_p(m_1, m'_1, m_2)(t) \mid$
 9 $t \in T_0\}$ is then a representative set of target VSUMs sampling
 10 the parameter space T such that every feasible program path of
 11 the implementation is considered.

12 The upcoming description of the concolic path exploration
 13 mechanism hence serves the goal to find a set of parameters that
 14 enumerates all feasible paths for the given CPR c_p and models
 15 m_1, m'_1, m_2 . Not all paths need be feasible, i.e., there may be
 16 unsatisfiable path conditions for the concrete input models.

4.3. Running Example

18 We now present a small running example of BrakeSystem-
 19 ControlSystem to exemplify our approach for the rest of this
 20 section. The source metamodel (BrakeSystem) and the target
 21 metamodel (ControlSystem) represent two engineering views.
 22 BrakeSystem has a set of BrakeDisc model elements with a
 23 diameter attribute, while ControlSystem has a correspond-
 24 ing set of AxleControlUnit model elements. The consistency
 25 relation CR specifies that each BrakeDisc has a correspond-
 26 ing AxleControlUnit whose controlProfile records the
 27 assigned drive mode, and whose absDecelThreshold repre-
 28 sents the target ABS deceleration limit for the axle unit and is
 29 computed as $diameter/50 \times multiplier$, where the multiplier is
 30 determined by a user input.

31 Adding a BrakeDisc triggers a CPR that reads
 32 $profileChoice \in [-1, 100]$ (denoted by pr): $pr < 0$
 33 skips unit creation; $0 \leq pr < 34$ sets off-road ($\times 0.70$);
 34 $34 \leq pr < 67$ sets comfort ($\times 0.85$); $67 \leq pr \leq 100$ sets
 35 sport ($\times 1.20$). In the formal terminology of Section 4.2, c_p is
 36 parameterized by $t = profileChoice \in T = [-1, 100]$, and
 37 the four intervals define four paths $\varphi^{(1)}, \dots, \varphi^{(4)}$ with effects
 38 $c_p^{(1)}, \dots, c_p^{(4)}$.

4.4. Symbolic Inputs and Symbolic Propagation

39 CoCoPath treats awaited user inputs as symbolic variables dur-
 40 ing execution. Conceptually, user inputs originate from interac-
 41 tion points in the consistency-preserving process and correspond
 42 to parameters $t \in T$ in the formalization of parameterized CPRs.
 43 For the purpose of systematic path exploration, CoCoPath rep-
 44 represents these user inputs as symbolic inputs, which range over
 45 an admissible finite or bounded domain and are associated with
 46 concrete runtime values before execution.

47 To propagate symbolic identities through the execution, Co-
 48 CoPath employs *dynamic taint tracking* on the JVM. In par-
 49 ticular, Galette is used to attach and propagate tags alongside
 50 concrete values, without altering program semantics. Import-
 51 tantly, taint tracking in CoCoPath is used *only* to propagate
 52 symbolic identity (and derived symbolic expressions) through
 53 data flow; it does not by itself prescribe how path constraints
 54 are constructed.

55 In our running example (Table 1), when the CPR reads
 56 `profileChoice` (denoted by `pr`), CoCoPath tags it as a sym-
 57 bolic variable at initialisation (Init row) and propagates this
 58 tag through all subsequent computations. When CPR takes any
 59 branch based on this input, CoCoPath records the corresponding
 60 inequality path constraint (PC).

4.5. Path Constraint Construction

62 During execution, CoCoPath records *path constraints* that char-
 63 acterize why a specific control-flow path was taken. CoCoPath
 64 supports two *complementary* mechanisms for constructing such
 65 constraints from executions. Both mechanisms operate on the
 66 same propagated symbolic information, but differ in *where* and
 67 *how* decision semantics are extracted.

68
 69 **Mechanism A: Bytecode-Level Constraint Extraction (Fully Au-
 70 tomatic).** In environments where bytecode-level interception
 71 is feasible, CoCoPath can derive constraints *automatically* from
 72 branch predicates at the JVM bytecode level. Concretely, de-
 73 cision points are identified by intercepting conditional branch
 74 instructions whose operands are tainted. The corresponding
 75 predicate is translated into a solver constraint and appended
 76 to the current path condition. This mechanism provides *full*
 77 *automation* of constraint construction and does not require CPR

1 modifications.

2 This mechanism requires that the bytecode instrumentation
 3 framework is compatible with the transformation runtime, in-
 4 cluding class loading and reflective execution patterns (e.g.,
 5 Open Services Gateway initiative (OSGi)).

6 **Mechanism B: CPR-Level Constraint Registration (Frame-
 7 work-Compatible).** For complex MDE runtimes where full
 8 bytecode-level interception is not yet reliable, CoCoPath sup-
 9 ports a CPR-level mechanism that makes decision semantics
 10 explicit at the level of the transformation logic. Whenever a
 11 CPR performs a decision that depends on a symbolic value (e.g.,
 12 a `switch` or conditional), the CPR invokes a standardized helper
 13 that registers the executed decision together with the symbolic
 14 value. The helper constructs the corresponding constraint (e.g.,
 15 $\ell = k$ for enumerated choices or $\ell \bowtie c$ for relational predicates,
 16 where $\bowtie \in \{<, >, \geq, \leq, \neq, =\}$ is the relational operator) and
 17 adds it to the current path condition.

18 This mechanism trades full automation for framework com-
 19 patibility while preserving systematic path exploration. In par-
 20 ticular, it avoids dependence on bytecode-level branch intercep-
 21 tion and remains compatible with complex execution stacks.
 22 In future work, automation could be increased by weaving in
 23 the standardized helper calls automatically, for example, when
 24 transforming a VITRUVIUS Reaction to its corresponding code.
 25 Still, as a limitation, this approach cannot handle decision points
 26 in third-party libraries.

27 **Relationship Between the Mechanisms** Mechanism A and
 28 Mechanism B are complementary: both aim to construct log-
 29 ically equivalent path conditions for decisions that depend on
 30 symbolic inputs. The difference lies in how decision points
 31 are detected—either implicitly via bytecode-level branch inter-
 32 ception (Mechanism A) or explicitly via CPR-level registration
 33 (Mechanism B). Which mechanism is employed is an imple-
 34 mentation choice and may depend on the characteristics of
 35 the transformation framework and runtime environment (e.g.,
 36 OSGi). This dual design allows CoCoPath to maximize au-
 37 tomation where bytecode-level interception is feasible, while
 38 retaining applicability to complex MDE frameworks through
 39 CPR-level constraint registration.

40 **Optional Domain Constraints (Solver-Side Refinement)** In ad-
 41 dition to path constraints derived from execution, CoCoPath can
 42 optionally incorporate *domain constraints*. Domain constraints
 43 restrict the admissible input domain based on external semantic
 44 knowledge (e.g., physical feasibility rules or application-specific
 45 configuration constraints). They do not affect the definition of
 46 paths and are not required for path completeness; rather, they
 47 serve as solver-side refinement to prune infeasible regions of
 48 the search space and improve performance.

49 In our running example (Table 1), the branch `if (pr < 0)`
 50 in $\varphi^{(1)}$ is the decision point based on the symbolic input pr . In
 51 Mechanism A, the interceptor would automatically detect this
 52 branch when the guard is evaluated against the tagged value
 53 and the path condition (PC) is recorded without any changes
 54 to the CPR. In Mechanism B, the CPR explicitly records the
 55 PC through the standardized helper at the branch point. In

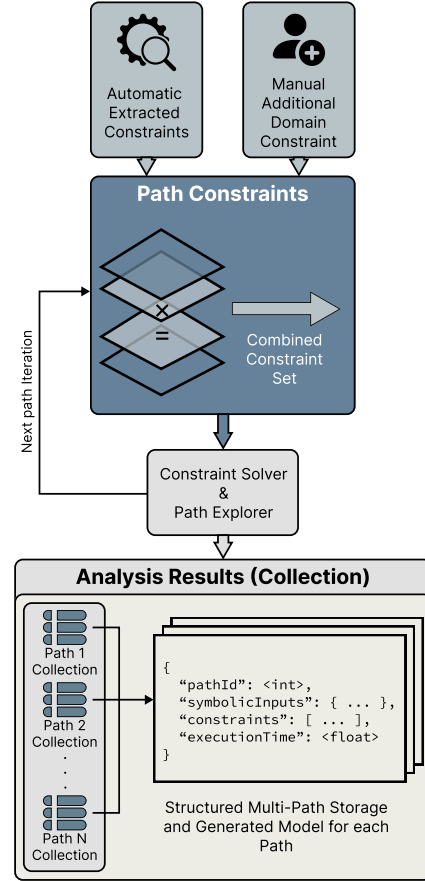


Figure 2 Systematic path exploration algorithm

56 both cases, the PC is the same: $PC_1 = \{pr < 0\}$ at the first
 57 execution with $pr = -1$, intersecting with T to obtain $[-1, 0)$,
 58 as shown in the PC and $PC \cap T$ columns. In the evaluation
 59 presented in this paper, admissible input domains are derived
 60 directly from the user interaction mechanism, and no additional
 61 domain constraints are required.

4.6. Systematic Path Exploration Loop

62 CoCoPath explores paths in an iterative concolic loop as shown
 63 in Figure 2. Starting from an initial concrete input assignment,
 64 the CPRs are executed concretely while symbolic identities are
 65 propagated and path constraints are collected. After each run,
 66 CoCoPath negates a selected decision constraint (or conjunction
 67 of constraints) and submits the resulting constraint system to
 68 an SMT solver to obtain a new satisfying assignment. If a satis-
 69 fying assignment exists, CoCoPath re-executes the CPRs with
 70 the new concrete inputs to explore a different path. The loop
 71 terminates when no further satisfying assignment can be gen-
 72 erated for unexplored path conditions. For each explored path,
 73 CoCoPath records the concrete inputs, the collected constraints,
 74 and the resulting repaired target model.

75 In our running example, Table 1 shows the full exploration.
 76 Starting with $pr = -1$, $\varphi^{(1)}$ records $PC_1 = \{pr < 0\}$ and
 77 the solver negates it to generate $pr = 0$. Successive executions
 78 cover $\varphi^{(2)} - \varphi^{(4)}$, with the solver generating $pr = 34$ and $pr =$
 79 67 from the Solver-Generated Input column. After $\varphi^{(4)}$, there
 80

Step	Input p	ControlProfile	Branch	Path Constraint (PC)	$PC \cap T$	Solver-Generated Input
Init	-1	—	—	—	$T = [-1, 100]$	—
$\varphi^{(1)}$	-1	skip	if ($pr < 0$)	$\{pr < 0\}$	$[-1, 0)$	$pr = 0$
$\varphi^{(2)}$	0	off-road	if ($pr \geq 0 \ \&\& \ pr < 34$)	$\{pr \geq 0, \ pr < 34\}$	$[0, 34)$	$pr = 34$
$\varphi^{(3)}$	34	comfort	if ($pr \geq 34 \ \&\& \ pr < 67$)	$\{pr \geq 34, \ pr < 67\}$	$[34, 67)$	$pr = 67$
$\varphi^{(4)}$	67	sport	else ($pr \geq 67$)	$\{pr \geq 67\}$	$[67, 100]$	—

Table 1 CoCoPath Path exploration for the running example ($pr = \text{profileChoice}$, bounded domain $T = [-1, 100]$).

1 are no more negations with unexplored paths, and the loop
2 terminates with target models corresponding to $c_p^{(1)}, \dots, c_p^{(4)}$.

3 The concrete realization of this loop in the VITRUVIUS-based
4 prototype is described in Section 5.

5. Prototypical CoCoPath Implementation

6 This section describes the prototypical implementation of Co-
7 CoPath and its integration with the VITRUVIUS framework, illus-
8 trated in Figure 3. The prototype instantiates the methodology
9 presented in Section 4, in particular the separation between
10 symbolic propagation and constraint construction. The CoCo-
11 Path approach is language-agnostic: it applies to any CPR that
12 can be represented as a program with branching control flow
13 (Section 4.2), provided a suitable dynamic taint-tracking tool is
14 available for the runtime. The current prototype is *Java-specific*
15 because it relies on Galette (Section 5.2), which instruments
16 JVM bytecode. In principle, other taint frameworks could serve
17 the same role for different languages. Pure symbolic execution
18 is a conceptual alternative, but is unlikely to scale to complex
19 MDE frameworks such as VITRUVIUS (Section 6.7). Addition-
20 ally, the current prototype targets CPRs implemented in *Java*
21 within the VITRUVIUS framework.

22 The current prototype realizes *CPR-level constraint registra-*
23 *tion* (Mechanism B) together with optional domain constraints.
24 Bytecode-level constraint extraction (Mechanism A) is not im-
25 plemented in the prototype for VITRUVIUS. This is due to prac-
26 tical limitations: VITRUVIUS relies on a modern Java 17 run-
27 time, extensive reflection, and OSGi-based class loading, which
28 currently prevents reliable interception of all control-flow deci-
29 sions at the bytecode level. We have validated the feasibility of
30 bytecode-level constraint extraction (Mechanism A) for plain
31 Java model transformations in a separate prototype based on
32 Galette, which can intercept native bytecode comparisons².
33 This indicates that the limitation is not inherent to Java 17,
34 but rather to the complexity of the VITRUVIUS execution stack.
35 However, this prototype is not evaluated in this paper.

36 Symbolic propagation is fully automated. User-decision
37 inputs are exposed by VITRUVIUS, tagged as symbolic prior to
38 execution, and propagated through the execution using dynamic
39 taint tracking with Galette. Constraint construction is localized
40 at explicit decision points in the CPR logic.

41 During execution, CoCoPath observes the concrete transfor-
42 mation runs, collects path constraints, and forwards them to the

43 constraint solver. By negating previously observed path condi-
44 tions, the constraint solver generates new concrete input assign-
45 ments, which are reinjected to systematically explore alternative
46 execution paths. For each execution, CoCoPath records the gen-
47 erated target model, the concrete input values, the collected path
48 constraints, and execution time, providing analyzable feedback
49 on alternative repair decisions.

50 The complete prototype, including its integration with
51 VITRUVIUS, Galette, and GreenSolver, is available as an open-
52 source repository (Jiang 2025a).

5.1. Model Transformation Framework -VITRUVIUS

53 We implement CoCoPath in the context of the VITRUVIUS frame-
54 work (Klare et al. 2021), which supports bidirectional and multi-
55 view consistency preservation based on rule-based model trans-
56 formations. In VITRUVIUS, consistency is restored by executing
57 the CPRs whenever a predefined trigger action was performed
58 and committed that necessitates propagating changes to other
59 models. VITRUVIUS serves as a concrete execution environment
60 in which CoCoPath is applied. User decisions required during
61 consistency preservation are exposed as user interaction input
62 variables and are treated as symbolic inputs. CoCoPath observes
63 the execution at runtime without interfering with the VITRUVIUS
64 transformation semantics.

5.2. Dynamic Taint Tracking with Galette

65 In CoCoPath, Galette (Hough & Bell 2025) is used to tag
66 user-decision inputs with symbolic identifiers before execution.
67 Galette is a dynamic taint tracking system for the JVM that
68 instruments bytecode at class load time. Its instrumentation in-
69 troduces a shadow state that enables the propagation of symbolic
70 metadata with concrete runtime values. As the transformation
71 logic is executed, Galette automatically propagates these tags
72 through method calls, field accesses, and control flow constructs.
73 Galette allows symbolic metadata to be tracked without modi-
74 fying the transformation logic and requiring language-specific
75 symbolic semantics.

5.3. CPR-Level Constraint Construction

76 In the current prototype, path constraints are constructed ex-
77 plicitly at the level of CPR implementations, corresponding to
78 Mechanism B introduced in Section 4.5. Symbolic informa-
79 tion is propagated automatically by Galette, while constraint
80 construction is triggered only at explicit decision points in the
81 transformation logic. Following Mechanism B (Section 4.5),
82 constraint construction is triggered at explicit decision points in
83 the transformation logic. Following Mechanism B (Section 4.5),
84 constraint construction is triggered at explicit decision points in
85

² <https://github.com/AnneKoziolek/galette-concolic-model-transformation/tree/comparison-interception-internal>

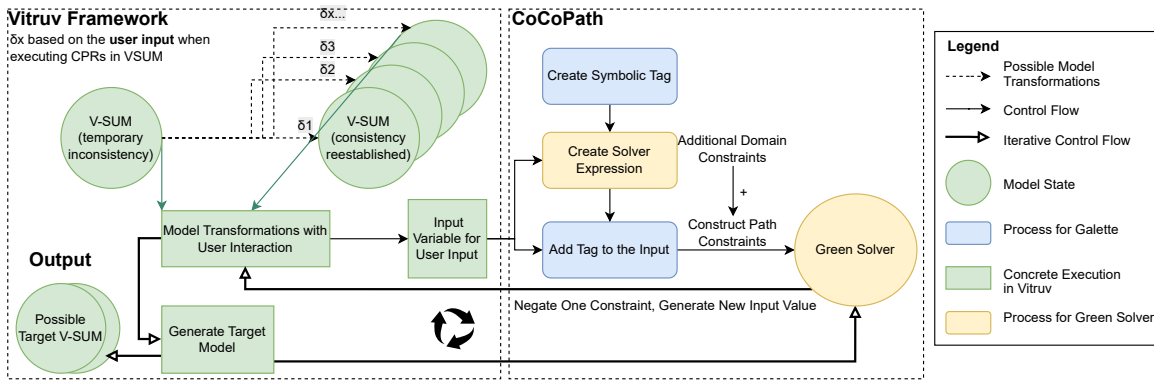


Figure 3 Prototypical CoCoPath implementation

1 the transformation logic. For binary predicates, the constructed
 2 constraint relates the symbolic variable ℓ to the compared constant using the predicate observed during execution. For multi-
 3 way control flow based on enumerated decisions, the selected
 4 branch yields an equality constraint of the form $\ell = k$, where
 5 k denotes the chosen option. This CPR-level approach local-
 6 izes constraint construction to semantically meaningful decision
 7 points and avoids reliance on bytecode-level branch interception.
 8 As a result, it remains compatible with the Java 17 runtime and
 9 the reflective execution patterns used by the VITRUVIUS frame-
 10 work. As a limitation, decisions occurring inside third-party
 11 libraries or framework-internal code are not captured by this
 12 mechanism.

13
 14 In future work, automation could be increased by weaving
 15 standardized constraint-registration calls into the generated Java
 16 code during the compilation of VITRUVIUS Reactions, while
 17 retaining the same conceptual approach.

18 5.4. Constraint Solving with the GreenSolver

19 The GreenSolver (Visser et al. 2012) is used in CoCoPath for
 20 transforming symbolic path and domain constraints into con-
 21 crete input values for subsequent executions. CoCoPath rep-
 22 represents both constraint types as symbolic expressions that are
 23 associated with tagged variables. The Solver collects these ex-
 24 pressions from the constructed constraint sets, translates them
 25 into Green’s internal representation, and delegates satisfiability
 26 checking to an SMT solver backend. In our current prototype,
 27 Green is configured to interface with the SMT solver Z3.

28 6. Evaluation

29 To address the RQs introduced in Section 1, we evaluate Co-
 30 CoPath through two case studies: Consistency preservation be-
 31 tween AMALTHEA and ASCET models based on an industrial
 32 case from Bosch and an extension of the running example.

33 We first describe the industrial context and the AMALTHEA-
 34 ASCET scenario that induces user-driven branching decisions
 35 (Section 6.1). We then explain how the scenario was integrated
 36 into our CoCoPath prototype (Section 6.3) and summarize the
 37 experimental setup used for all measurements (Section 6.4).
 38 We then present our results for RQ1 (Section 6.5), RQ2 (Sec-

tion 6.6), RQ3 (Section 6.7), and RQ4 (Section 6.8). We discuss
 threats to validity in Section 6.9.

41 6.1. AMALTHEA-ASCET case study (A² case)

42 **Industrial Context.** The evaluated scenario is drawn from an
 43 industrial, model-driven development setting at Bosch, where
 44 development incorporates the tool suites ASCET (Lefarth et
 45 al. 1998; ASCET Repository 2024) and AMALTHEA (now
 46 APP4MC). While both tool suites support model-driven devel-
 47 opment of embedded systems, they focus on different concerns
 48 and are therefore used in parallel by different engineering teams.

49 In the examined setting, Electronic Control Units (ECUs) are
 50 developed concurrently using AMALTHEA and ASCET mod-
 51 els. AMALTHEA is used to model the ECU architecture and
 52 its components, targeting operating-system and hardware-level
 53 aspects. In contrast, ASCET is used to specify the functional
 54 behavior of components, largely independent of the target op-
 55 erating system. Both AMALTHEA and ASCET generate C
 56 code, which is subsequently integrated for deployment on the
 57 ECU. As a result, there is a semantic overlap between the mod-
 58 els, for example requiring that architectural elements defined in
 59 AMALTHEA correspond to elements in ASCET.

60 **A² case** We evaluate CoCoPath using an adapted consistency-
 61 preservation scenario from the automotive domain described
 62 by Mazkatli (2016). The evaluated CPRs are triggered when a
 63 new Task is added to an AMALTHEA model and require the
 64 creation of a corresponding task element in the ASCET model.

65 As illustrated in Figure 4, this mapping is not unique.
 66 While AMALTHEA distinguishes tasks only at an abstract level,
 67 ASCET defines multiple concrete task subtypes, in-
 68 cluding InitTask, PeriodicTask, SoftwareTask, and
 69 TimeTableTask. Selecting the appropriate target-side subtype
 70 depends on domain knowledge and therefore cannot be resolved
 71 automatically.

72 Consequently, the corresponding CPR contains an explicit
 73 decision point that requires user input to select among these
 74 alternatives. Different choices lead to structurally different
 75 ASCET models: some task types introduce additional attributes
 76 and references (e.g., PeriodicTask), while other options re-
 77 sult in minimal or even no changes to the target model. As a

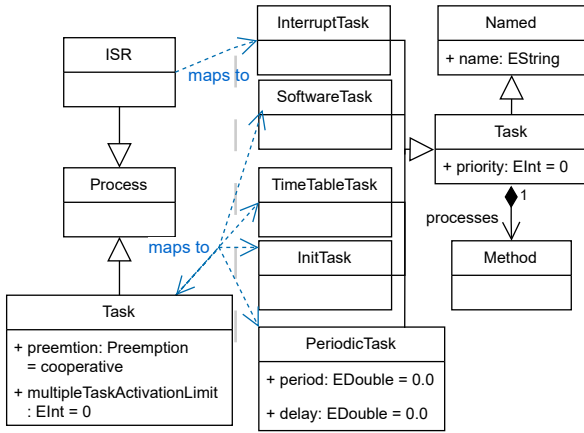


Figure 4 Snippet of the AMALTHEA metamodel (left), the ASCET metamodel (right), and the mapping relations (blue), based on a simplified scenario in Mazkatli (2016).

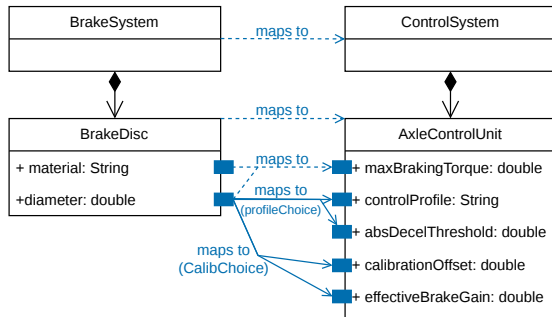


Figure 5 Snippet of the BrakeSystem metamodel (left), the ControlSystem metamodel (right), and the mapping relations (blue). Solid arrows: mappings requiring user input.

67–100 Track (+0.5). This results in `effectiveBrakeGain` calculated as `effectiveBrakeGain = absDecelThreshold + calibrationOffset`. If the first action skips unit creation, the second finds no matching element and is a no-op.

6.3. Prototype Integration

To apply CoCoPath to the case studies described in Section 6.1 and 6.2, we integrated the relevant parts into the VITRUVIUS framework. Concretely, we (i) converted the involved meta-model excerpts to Ecore models and (ii) implemented the CPRs described above in VITRUVIUS’s Reactions language. Our implementation propagates changes from AMALTHEA to ASCET and from BrakeSystem to ControlSystem, as this is the only direction required for the evaluated scenario. The implementation is accessible at Jiang (2025b) and Jiang (2026), respectively.

To execute the A^2 case, we add AMALTHEA task(s) and trigger CoCoPath to explore alternative repairs that restore consistency in the VSUM. The CPR that reacts to adding a Task in AMALTHEA contains an explicit decision point implemented as a switch-case statement. The user provides a numerical choice from “0” to “4”, which determines the mapping for the newly created task in the ASCET model: “0” creates an InitTask, “1” a PeriodicTask, “2” a SoftwareTask, “3” a TimeTableTask, and “4” delays the decision (i.e., does not create a corresponding ASCET task and keeps the inconsistency). For the evaluation, we consider two scenarios to study CoCoPath’s behavior under increasing parameter-space size: (i) adding one AMALTHEA task, resulting in a single symbolic input variable with five possible values, and (ii) adding two different AMALTHEA tasks, resulting in two independent symbolic input variables and 25 input combinations.

To execute the BCCS case, we likewise implemented the metamodels as Ecore models and the CPRs in VITRUVIUS’s Reactions language with two decision points. The first decision point implemented as a conditional branch on `profileChoice`, which is partitioned into four intervals determining whether and how an `AxleControlUnit` is created. The second decision point relies on `calibChoice`: a value partitioned into three calibration levels that sets the `calibrationOffset` and `effectiveBrakeGain`. For the evaluation, we consider one scenario in which one `BrakeDisc` is added, yielding two symbolic input variables (`profileChoice` and `calibChoice`).

6.4. Experimental Setup.

All experiments were conducted on a workstation equipped with an Intel Core i7-9750H processor (6 cores, 2.6–4.5 GHz), 16 GB of DDR4 memory, and NVMe SSD storage, running Windows 10 (64-bit, Build 19045). We used OpenJDK Temurin version 17.0.2 with JVM options `-Xmx4G -Xms1G -ea`. The setup was based on Eclipse Modeling Framework (EMF) version 2.28, and constraint solving was performed using the Z3 SMT solver via the GreenSolver framework.

6.5. RQ1: Path Coverage and Outcome Analysis

To address RQ1, we conducted controlled experiments on two case studies. All model differences were quantified using EMF Compare (Brun & Pierantonio 2008).

1 result, the impact of a decision is not obvious without executing
2 the transformation, making this scenario well-suited to evalu-
3 ate CoCoPath’s systematic exploration of decision-dependent
4 execution paths.

5 6.2. BrakeDisc-ControlSystem case study (BCCS case)

6 We created an open case study to assess generalizability, as in-
7 dustrial artefacts are usually confidential. It evaluates CoCoPath
8 on *continuous inputs* whose path condition is ranges partitioned
9 by inequalities instead of discrete enumerations. The case study
10 is an extension of the running example in Section 4.3. Figure 5
11 shows the two metamodels and their mapping relations. When
12 a `BrakeDisc` is inserted, a CPR with two actions is triggered.
13 The first reads `profileChoice` $\in [-1, 100]$: -1 skip unit cre-
14 ation, while $0-33$, $34-66$, and $67-100$ assign `controlProfile`
15 (the drive mode) to Off-road, Comfort, and Sport, respectively.
16 Each mode applies an aggressiveness multiplier (0.70, 0.85,
17 1.20) to compute `absDecelThreshold` = `diameter/50` \times
18 `multiplier`. `absDecelThreshold` represents the ABS decel-
19 eration limit for the unit. The second reads `calibChoice`
20 $\in [0, 100]$ and sets `calibrationOffset` (a fine-tuning correc-
21 tion). $0-32$ is Conservative (-0.5), $33-66$ Standard (0.0), and

Path ID	user_choice	A	At	R	D	Total
0	0	1	1	1	0	3
1	1	1	3	1	0	5
2	2	1	1	1	0	3
3	3	1	1	1	0	3
4	4	0	0	0	0	0

Legend: A = Add Elements, At = Change Attributes, R = Change Reference, D = Delete Elements, Total = Total Atomic Differences

Table 2 Single-enumerate exploration: outcome statistics.

6.5.1. A^2 case: Single-Enumerate Variable Exploration

A single symbolic parameter `user_choice` was explored, which ranges over a finite parameter space $T = \{0, 1, 2, 3, 4\}$. CoCoPath automatically explored all possible paths by iteratively negating path constraints of the form $user_choice = v$, where $v \in T$. CoCoPath reached **complete coverage** (5/5 paths). Each explored path produced a concrete model transformation result that could then be evaluated using EMF Compare metrics. Table 2 summarizes the number of added elements (A), changed attributes (At), changed references (R), deleted elements (D), and the resulting total atomic changes, computed as $Total = A + At + R + D$. The results show that the transformation is a no-op transformation when $user_choice = 4$, i.e., the option that does not create a new task; and the most significant structural change occurs when $user_choice = 1$, i.e., when a new `PeriodicTask` is created.

6.5.2. A^2 case: Multi-Enumerate Variable Exploration

We extend the experiment to two symbolic variables ($user_choice_1, user_choice_2$) $\in T \times T$, where $T = \{0, 1, 2, 3, 4\}$. This gives a Cartesian product of 25 decision combinations. CoCoPath achieved **complete coverage** (25/25 paths). The result of the EMF Compare metrics is shown in Table 3. The total difference is calculated in the same way as the single-variable exploration. The systematic exploration of the parameter space yielded results in fully analyzable outcomes. Although the numerical values of model changes are specific to this case study, the form of the results shown indicates the practical value of CoCoPath: First, the complete path exploration allows engineers to immediately see the possible model outcomes that arise from different user inputs, without manually running 25 transformation executions. Second, the calculated model difference metrics provide actionable guidance: combinations with significant total differences signal decision points with high impact on the resulting model, while combinations with near-zero differences identify regions of the parameter space where engineers can delay decisions or choose with defaults. Finally, because CoCoPath reports these results together with the associated path constraints, the exploration outcomes can be linked back to the transformation logic that triggered each model variant. In the evaluated scenario, this makes explicit that choosing a `PeriodicTask` consistently leads to the largest structural changes, while delaying the decision yields a no-op transformation. Such insights allow engineers to quickly identify high-impact and low-impact choices without manual experimentation.

6.5.3. BCCS case: Multi-Continuous Variable Exploration

Unlike the A^2 case, in which path constraints are expressed as value equalities, the BCCS case engages another qualitative aspect of the exploration engine: CoCoPath negates interval constraints of the form, arising from the inequality partitions over $profileChoice \in [-1, 100]$ and $calibChoice \in [0, 100]$. CoCoPath achieved **complete coverage (10/10 paths)**. The skip branch ($profileChoice < 0$) generates one path in which no `AxleControlUnit` is created; the remaining nine paths are generated by the Cartesian product of three intervals for `profileChoice` and three intervals for `calibChoice`, for a total of $1 + 3 \times 3 = 10$ paths. Table 4 lists the EMF Compare metrics. The skip path is a structural no-op, while all nine non-skip paths are structurally identical. Each produces one `AxleControlUnit` with five attributes set and one containment reference added.

Answer to RQ1

CoCoPath achieves a complete and systematic exploration of all possible paths induced by user inputs with respect to the evaluated consistency-preserving transformation paths. The obtained outputs are analyzable: First, EMF Compare metrics provide a structured and numeric characterization for each transformation outcome. Engineers can thus compare paths, estimate the relative impact of different user inputs, and interaction effects among symbolic inputs. The analysis provides actionable feedback, such as showing the highest-impact input combinations and regions of no-op in the parameter space, and quantifying the consequences of delaying or altering user inputs. Second, CoCoPath provides the path constraint and output models for each path that engineers can inspect.

6.6. RQ2: Performance and Scalability

To answer RQ2, we evaluate the scaling efficiency of CoCoPath's runtime performance as the number of symbolic variables increases, which in turn leads to a larger decision space and more possible execution paths, and we reason about the theoretical complexity of path exploration. For the sake of space, we only present the A^2 case results here. The results for BCCS case can be found on GitHub (Jiang 2025a).

In order to assess scalability independently of initialization time, we analyze the average execution time per path. As presented in Table 5, execution time in the multi-variable scenario increases to 49.0ms on average, compared to 34.4ms in the single-variable scenario, indicating a rise of only $1.42 \times$ despite a $5 \times$ increase in explorations of paths. Overall, the results indicate that initialization overhead dominates short executions but does not scale with the number of explored paths. Once this one-time cost is amortized, the execution time grows with the number of feasible paths and therefore scales fairly well for multiple-variable path exploration.

Theoretically, the complexity of CoCoPath can be decomposed into (1) the number of feasible execution paths and (2) the cost of exploring each path. Let d be the number of symbolic decision points and b_i the branching factor of decision point i . In the worst case, the number of explored paths is bounded by

Path ID	c1	c2	A	D	At	R	Total	Path ID	c1	c2	A	D	At	R	Total	Path ID	c1	c2	A	D	At	R	Total
0	0	0	2	0	2	2	6	8	1	3	2	0	4	2	8	16	3	1	2	0	4	2	8
1	0	1	3	0	3	2	8	9	1	4	1	0	3	1	5	17	3	2	2	0	2	2	6
2	0	2	2	0	2	2	6	10	2	0	2	0	2	2	6	18	3	3	2	0	2	2	6
3	0	3	2	0	2	2	6	11	2	1	2	0	4	2	8	19	3	4	1	0	1	1	3
4	0	4	1	0	1	1	3	12	2	2	2	0	2	2	6	20	4	0	1	0	1	1	3
5	1	0	2	0	4	2	8	13	2	3	2	0	2	2	6	21	4	1	1	0	3	1	5
6	1	1	2	0	6	2	10	14	2	4	1	0	1	1	3	22	4	2	1	0	1	1	3
7	1	2	2	0	4	2	8	15	3	0	2	0	2	2	6	23	4	3	1	0	1	1	3
																24	4	4	0	0	0	0	0

Legend: c1/c2 = user choices, A/D = Added / removed elements, At / R = Changed attributes / references, Tot = Total atomic differences.

Table 3 Multi-enumerate exploration: outcome statistics for all 25 combinations of user_choice_1 and user_choice_2.

Path	profileChoice	calibChoice	A	At	R	D	Total
0	-1	-	0	0	0	0	0
1	0-33	0-33	1	5	1	0	7
2	0-33	34-66	1	5	1	0	7
3	0-33	67-100	1	5	1	0	7
4	34-66	0-33	1	5	1	0	7
5	34-66	34-66	1	5	1	0	7
6	34-66	67-100	1	5	1	0	7
7	67-100	0-33	1	5	1	0	7
8	67-100	34-66	1	5	1	0	7
9	67-100	67-100	1	5	1	0	7

Legend: A / D = Added / Removed elements, At / R = changed attributes / references, Total = total atomic differences.

Table 4 Multi-continuous exploration: outcome statistics.

Scenario	Paths	Init.	Avg./Path	Total
Single-variable	5	962	34.4	1134
Multi-variable	25	1873	49.0	3099

Table 5 Execution time summary (ms)

1 $\prod_{i=1}^d b_i$, corresponding to the classical path-explosion problem
2 known from symbolic execution (Cadaru & Sen 2013). For each
3 path, CoCoPath performs one concrete execution of the CPR
4 chain and one SMT solver query generated from the negated
5 path condition. The solver cost depends on the theory of the path
6 constraints. In the current prototype, constraints consist only of
7 linear equalities and inequalities, which can be solved efficiently
8 even for many variables. More expressive constraint theories
9 such as bitvectors or strings are also supported by modern SMT
10 solvers, although their solving complexity can be higher in the
11 worst case (De Moura & Bjørner 2008). In the CoCoPath set-
12 ting, however, path constraints typically involve only a small
13 number of symbolic variables originating from user inputs, so
14 we expect feasible solver cost even when additional theories
15 are used. Furthermore, CoCoPath explores only feasible paths;
16 infeasible path conditions are pruned automatically by the SMT
17 solver. Since exploration starts from a concrete model state
18 and a specific change and assuming VSUMs with low coupling
19 and high cohesion, we expect the number of feasible paths to
20 typically be much smaller than the theoretical worst-case bound

even in large VSUMs. Empirically exploring the scalability in
real-world use cases with larger VSUMs will be future work.

Answer to RQ2

CoCoPath demonstrates good runtime performance and scal-
ability for multi-variable path exploration. Despite the frame-
work initialization accounting for the majority of the total run-
time for small workloads, the amortized per-path execution
cost is relatively stable as the decision space increases, en-
abling efficient and systematic exploration of multi-variable
execution paths.

6.7. RQ3: Applicability

To answer RQ3, we compare CoCoPath with representative
state-of-the-art symbolic execution tools with respect to their ap-
plicability to VITRUVIUS frameworks. We select Java PathFinder
(Păsăreanu & Rungta 2010), CATG (Tanno et al. 2015), and
JDart (Luckow et al. 2016). These tools are used extensively
in research on symbolic execution techniques. JPF is a model-
checking-based symbolic execution tool, CATG is a concolic
testing tool with systematic path exploration, and JDart provides
dynamic symbolic execution for Java bytecode.

Table 6 summarizes the compatibility of the evaluated tools
with key technologies required by VITRUVIUS modeling frame-
works. VITRUVIUS relies on EMF-based models, extensive re-
flection, and dynamic class loading, and execution within an
OSGi runtime environment. The results reveal that existing sym-
bolic execution tools are not directly applicable to VITRUVIUS-
based MDE workflows. In particular, for these tools, there is
a need for customized virtual machines or restricted execution
environments, which are not compatible with EMF and OSGi-
based systems. In contrast, CoCoPath operates on standard
JVM bytecode and maintains full compatibility with VITRUVIUS,
EMF, and OSGi. Because of these fundamental incompatibili-
ties, it is impossible to compare CoCoPath directly with existing
tools at the workload level.

Answer to RQ3

To the best of our knowledge, CoCoPath is the only evaluated
symbolic execution approach that is fully compatible with
VITRUVIUS, EMF, and OSGi. workflows.

Table 6 Framework compatibility comparison

Tool	VITRUVIUS	EMF	OSGi
Java PathFinder	✗	✗	✗
CATG	✗	✗	✗
JDart	✗	✗	✗
CoCoPath	✓	✓	✓

Table 7 Peak heap memory usage and memory overhead.

Configuration	Peak Heap (MB)	OV _{memory}
Baseline (Pure VITRUVIUS)	14	1.00
Galette-only	17	1.21
Full CoCoPath	19	1.36

6.8. RQ4: Memory Overhead

This research question explores the memory overhead introduced by CoCoPath. We focus on peak heap memory usage and quantify overhead. Memory consumption is measured based on peak heap usage during execution. All configurations are evaluated under identical workloads and execution environments, and the resulting peak values are stable across repeated experiments. Memory overhead is defined as:

$$OV_{\text{memory}} = \frac{\text{Peak Heap}_{\text{Configuration}}}{\text{Peak Heap}_{\text{Baseline}}}$$

An overhead of $1.0\times$ means there was no additional memory consumption compared to the baseline execution. In summary, table 7 displays the measured peak heap usage and associated memory overhead for all configurations.

Introducing dynamic taint tracking alone (Galette-only) increases peak heap usage from 14 MB to 17 MB, denoted by a memory overhead of 1.21. The peak heap usage is further increased by enabling full CoCoPath functionality to 19 MB as it leads to a memory overhead of 1.36.

A moderate additional memory overhead can be observed. For all experiments, the additional retained memory per explored path is still in the order of a few kilobytes. This leads to a stable peak heap usage regardless of the increase in the number of explored paths, and the memory overhead mainly depends on the one-time initialization.

Answer to RQ4

Galette-only execution results in a memory overhead of **1.21**, while full CoCoPath execution results in an overhead of **1.36**. The majority of the memory overhead comes from the dynamic taint tracking. In general, CoCoPath has low and constant memory overhead.

6.9. Threats to Validity

Construct Validity. For RQ4, we measure memory overhead using peak heap usage. Peak memory usage indicates worst-case memory pressure but does not capture fine-grained object lifetimes. Despite that, it is adequate for comparing configurations under identical workloads.

Internal Validity. All experiments were run on the same hardware and software configuration, using identical input models and transformation logic across all configurations. The experiments were run multiple times (five runs per configuration) to obtain stable results and avoid the risk of measurement noise. However, factors such as JVM warm-up and garbage collection behavior could affect the measurements.

External Validity. Our evaluation focuses on a single but representative case study from the industry involving consistency preservation between AMALTHEA and ASCET models. Although this case study considers real aspects of the MDE processes used in industry, including user decisions and heterogeneous metamodels, the obtained results may not generalize to all transformations or model sizes. Further studies on additional case studies and domains are required to assess the general applicability of CoCoPath.

7. Discussion

The prototypical implementation of CoCoPath demonstrates the feasibility of combining dynamic symbolic execution techniques with model transformation frameworks to systematically explore consistency-preserving paths in VSUMs. One of the key strengths of the prototype lies in its non-intrusive integration with VITRUVIUS. Therefore, the concept can be integrated with other consistency preservation approaches or also model transformation approaches in general. Moreover, since CoCoPath prototyped implementation is based on standard JVM instrumentation, it is possible to generalize this implementation to any Java-based model transformation framework. CoCoPath observes transformation executions without altering their semantics, ensuring compatibility with existing CPRs and transformation logic. Combined with SMT Solver, CoCoPath can iteratively generate alternative user decisions, enabling systematic exploration of possible target VSUM states. This supports developers in making informed decisions and reveals itself, e.g., in the form of an aid in estimating the resulting model drift, which can be used to determine how quickly the inconsistency should be repaired. Despite these strengths, the prototype has several limitations. First, the reliance on runtime observation introduces overhead, which may become significant for large-scale transformations with numerous decision points. Second, the tag-based metadata extraction currently only works for comparisons in CPR implementation. It is not possible to track comparisons in arbitrary Java code. Finally, while domain constraints can be manually specified to restrict the search space, this process requires expert knowledge and may limit automation. We are also considering related approaches like (Eramo et al. 2015) to represent the consistency-preserving paths in a shorter way. Moreover, developing mechanisms to help navigate

1 and compare the generated target models, e.g., based on ranking
2 according to model drift, is another future direction.

3 8. Conclusion

4 This paper presents CoCoPath, a concolic execution framework
5 for systematically exploring execution paths in consistency-
6 preserving model transformations. Our approach combines
7 dynamic taint tracking with constraint solving to automatically
8 explore all feasible paths induced by user decisions, generating
9 corresponding target VSUMs and considering path constraint
10 sets for each path. Through two case studies, we validated
11 that CoCoPath achieves complete path coverage and produces
12 analyzable transformation results, while demonstrating good
13 performance and scalability in multi-variable path exploration.
14 CoCoPath provides engineers with the capability to quantita-
15 tively analyze the impact of decisions, enabling more informed
16 decision-making when facing temporary inconsistency. Future
17 work includes applying CoCoPath to larger-scale cases
18 to evaluate its applicability, scalability on deeper CPR chains
19 and non-linear constraint types, and exploring the potential of
20 CoCoPath in general model transformations.

21 Acknowledgments

22 This work was funded by the Deutsche Forschungsgemeinschaft
23 (DFG) - SFB 1608 - 501798263 and by the Topic Engineering
24 Secure Systems of the Helmholtz Association (HGF) and sup-
25 ported by KASTEL Security Research Labs, Karlsruhe.

26 References

27 *ASCET Repository*. (2024). Retrieved 2025-12-12, from [https://](https://github.com/etas/ascet)
28 github.com/etas/ascet
29 Atkinson, C., Stoll, D., & Bostan, P. (2008). Orthographic
30 software modeling: a practical approach to view-based devel-
31 opment. In *International Conference on Evaluation of Novel*
32 *Approaches to Software Engineering* (pp. 206–219).
33 Bell, J., & Kaiser, G. (2014). Phosphor: illuminating dynamic
34 data flow in commodity JVMs. In *Proceedings of the 2014*
35 *ACM International Conference on Object Oriented Program-*
36 *ming Systems Languages and Applications* (p. 83–101). New
37 York, NY, USA: ACM. doi: 10.1145/2660193.2660212
38 Brun, C., & Pierantonio, A. (2008). Model differences in the
39 eclipse modeling framework. *UPGRADE, The European*
40 *Journal for the Informatics Professional*, 9(2), 29–34.
41 Bruneliere, H., Burger, E., Cabot, J., & Wimmer, M. (2019). A
42 feature-based survey of model view approaches. *Software &*
43 *Systems Modeling*, 18(3), 1931–1952.
44 Cadar, C., & Sen, K. (2013). Symbolic execution for software
45 testing: three decades later. *C. of the ACM*, 56(2), 82–90.
46 Cicchetti, A., Ciccozzi, F., & Pierantonio, A. (2019). Multi-view
47 approaches for software and system modelling: a systematic
48 literature review. *SoSyM*, 18(6), 3207–3233.
49 De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT
50 solver. In *International conference on Tools and Algorithms*
51 *for the Construction and Analysis of Systems* (pp. 337–340).

Demuth, A., Lopez-Herrejon, R. E., & Egyed, A. (2015). 52
Constraint-driven modeling through transformation. *Soft-* 53
ware & Systems Modeling, 14(2), 573–596. 54
Demuth, A., Riedl-Ehrenleitner, M., Nöhrer, A., Hehenberger, 55
P., Zeman, K., & Egyed, A. (2015). DesignSpace: an infras- 56
tructure for multi-user/multi-tool engineering. In *Proceedings* 57
of the 30th Annual ACM SAC (pp. 1486–1491). 58
Egyed, A., Demuth, A., Ghabi, A., Lopez-Herrejon, R., Mäder, 59
P., Nöhrer, A., & Reder, A. (2011). Fine-tuning model 60
transformation: Change propagation in context of consistency, 61
completeness, and human guidance. In *Intl. Conf. on Theory* 62
and Practice of Model Transformations (pp. 1–14). 63
Elaasar, M., Rouquette, N., Wagner, D., Oakes, B. J., Hamou- 64
Lhadj, A., & Hamdaqa, M. (2023). openCAESAR: Bal- 65
ancing Agility and Rigor in Model-Based Systems Engi- 66
neering. In *2023 ACM/IEEE MODELS-C*. doi: 10.1109/ 67
MODELS-C59198.2023.00051 68
Eramo, R., Pierantonio, A., & Rosa, G. (2015). Managing uncer- 69
tainty in bidirectional model transformations. In *Proceedings* 70
of the 2015 ACM SIGPLAN International Conference on 71
Software Language Engineering (pp. 49–58). 72
Famelis, M., Salay, R., Di Sandro, A., & Chechik, M. (2013). 73
Transformation of models containing uncertainty. In *Intl.* 74
conf. on Model-Driven Engineering Languages and Systems 75
(pp. 673–689). 76
Godefroid, P., Klarlund, N., & Sen, K. (2005, June). DART: 77
directed automated random testing. *SIGPLAN Not.*, 40(6), 78
213–223. Retrieved from [https://doi.org/10.1145/1064978](https://doi.org/10.1145/1064978.1065036)
1065036 doi: 10.1145/1064978.1065036 79
80
Golra, F. R., Beugnard, A., Dagnat, F., Guerin, S., & Guy- 81
chard, C. (2016, March). Addressing Modularity for Het- 82
erogeneous Multi-Model Systems Using Model Federation. 83
In *Companion Proceedings of the 15th International Con-* 84
ference on Modularity. New York, NY, USA: ACM. doi: 85
10.1145/2892664.2892701 86
Hough, K., & Bell, J. (2025). Dynamic Taint Tracking for 87
Modern Java Virtual Machines. *Proceedings of the ACM on* 88
Software Engineering, 2(FSE), 1757–1779. 89
Jiang, B. (2025a). *CoCoPath: Concolic Path Exploration* 90
for Consistency-Preserving Model Transformations. [https://](https://github.com/IngridJiang/CocoPath)
91 github.com/IngridJiang/CocoPath. 92
Jiang, B. (2025b). *Implementation of the AMALTHEA-ASCET* 93
synchronization case study in VITRUVIUS. [https://github](https://github.com/IngridJiang/Amalthea-acset)
94 [.com/IngridJiang/Amalthea-acset](https://github.com/IngridJiang/Amalthea-acset). 95
Jiang, B. (2026). *Implementation of the BrakeSystem-* 96
ControlSystem synchronization case study in VITRUVIUS.
97 <https://github.com/IngridJiang/BrakeDiscwithUserinput>. 98
Kelly, S., Lyytinen, K., & Rossi, M. (1996). Metaedit+ a fully 99
configurable multi-user and multi-tool case and came environ- 100
ment. In *International Conference on Advanced Information* 101
Systems Engineering (pp. 1–21). 102
Kelly, S., & Tolvanen, J.-P. (2021). Collaborative modelling and 103
metamodelling with MetaEdit+. In *2021 ACM/IEEE Interna-* 104
tional Conference on Model Driven Engineering Languages 105
and Systems Companion (MODELS-C) (pp. 27–34). 106
Klare, H., Kramer, M. E., Langhammer, M., Werle, D., Burger, 107
E., & Reussner, R. (2021). Enabling consistency in view- 108

- 1 based system development - The Vitruvius approach. *Journal of Systems and Software*, 171, 110815.
- 2
- 3 Kretschmer, R., Khelladi, D. E., Lopez-Herrejon, R. E., & Egyed, A. (2021). Consistent change propagation within models. *Software and Systems Modeling*, 20(2), 539–555.
- 4
- 5 Kukucka, J., Pina, L., Ammann, P., & Bell, J. (2022). Confetti: Amplifying concolic guidance for fuzzers. In *Proceedings of the 44th Intl. Conf. on Software Engineering* (pp. 438–450).
- 6
- 7 Laghouaouta, Y., & Laforcade, P. (2020). Dealing with uncertainty in model transformations. In *Proceedings of the 35th Annual ACM SAC* (pp. 1595–1603).
- 8
- 9 Lefarth, U., Baum, U., Beck, T., Werther, K., & Zurawka, T. (1998). An integrated approach to rapid product development for embedded automotive control systems. *Control Engineering Practice*, 6(4), 529–540.
- 10
- 11 Lucas, F. J., Molina, F., & Toval, A. (2009). A systematic review of UML model consistency management. *Information and Software technology*, 51(12), 1631–1645.
- 12
- 13 Luckow, K., Dimjašević, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., ... Raman, V. (2016). JDart: A dynamic symbolic analysis framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 442–459).
- 14
- 15 Mazkatli, M. (2016). *Consistency Preservation in the Development Process of Automotive Software* (Master's thesis). (ISBN: 9781543532913) doi: 10.5445/IR/1000143392
- 16
- 17 Mazkatli, M., Burger, E., Koziolok, A., & Reussner, R. H. (2017). Automotive systems modelling with Vitruvius. In *INFORMATIK 2017* (pp. 1487–1498).
- 18
- 19 Meier, J., Klare, H., Tunjic, C., Atkinson, C., Burger, E., Reussner, R. H., & Winter, A. (2019). Single Underlying Models for Projectional, Multi-View Environments. In *MODEL-SWARD* (pp. 117–128).
- 20
- 21 Ouyang, Y., Shao, K., Chen, K., Shen, R., Chen, C., Xu, M., ... Zhang, L. (2023). MirrorTaint: Practical Non-Intrusive Dynamic Taint Tracking for JVM-Based Microservice Systems. In (p. 2514–2526). IEEE Press. doi: 10.1109/ICSE48619.2023.00210
- 22
- 23 Păsăreanu, C. S., & Rungta, N. (2010). Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering* (pp. 179–180).
- 24
- 25 Pascual, R., Beckert, B., Ulbrich, M., Kirsten, M., & Pfeifer, W. (2024). Formal Foundations of Consistency in Model-Driven Development. In T. Margaria & B. Steffen (Eds.), *12th Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2024)* (Vol. 15222, pp. 178–200). Springer. doi: 10.1007/978-3-031-75380-0_11
- 26
- 27 Stünkel, P., König, H., Lamo, Y., & Rutle, A. (2021, December). Comprehensive Systems: A Formal Foundation for Multi-Model Consistency Management. *Formal Aspects of Computing*, 33(6). doi: 10.1007/s00165-021-00555-2
- 28
- 29 Tanno, H., Zhang, X., Hoshino, T., & Sen, K. (2015). TesMa and CATG: automated test generation tools for models of enterprise applications. In *37th IEEE/ACM International Conference on Software Engineering* (Vol. 2, pp. 717–720).
- 30
- 31 Ubukata, T., Zhang, M., Yamauchi, T., Li, J., & Tei, K. (2025). Graph-contextual reinforcement learning for efficient exploration in directed controller synthesis. In *25th international conference on software quality, reliability, and security, QRS 2025 - companion, hangzhou, china, july 16-20, 2025* (pp. 777–779). IEEE. Retrieved from <https://doi.org/10.1109/QRS-C65679.2025.00104> doi: 10.1109/QRS-C65679.2025.00104
- 32
- 33 Visser, W., Geldenhuys, J., & Dwyer, M. B. (2012). Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th Intl. Symposium on the Foundations of Software Engineering* (pp. 1–11).
- 34
- 35 Zhang, W., Zhang, W., Strüber, D., & Hebig, R. (2025). An empirical study of manual abstraction between class diagrams and code of open-source systems. *Software and Systems Modeling*, 1–27.
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56

About the authors

Bowen Jiang is a PhD researcher at Karlsruhe Institute of Technology (KIT), Germany. She received her M.Sc. from WASEDA University, Japan in 2024. Her research interests include MDE, software testing, and AI4SE. You can contact the author at bowen.jiang@kit.edu or visit https://mcse.kastel.kit.edu/staff_bowen_jiang.php.

Miriam Boss is a PhD researcher at KIT. She received her M.Sc. from RWTH Aachen University in 2025. Her research interests include MDE, human-in-the-loop inconsistency management, and software language engineering. You can contact the author at boss@kit.edu or visit https://dsis.kastel.kit.edu/staff_miriam_boss.php.

Thomas Weber is a PhD researcher at KIT. He received his M.Sc. from KIT in 2023. His research interests include MDE, model consistency preservation, and intellectual property protection in MDE. You can contact the author at thomas.weber@kit.edu or visit https://dsis.kastel.kit.edu/staff_Thomas_Weber.php.

Weixing Zhang is a PostDoc at KIT. He received his PhD at the University of Gothenburg. His research interests include SE, Empirical SE, AI4SE. You can contact the author at weixing.zhang@kit.edu or visit <https://wilson008.github.io/>.

Mattias Ulbrich is a postdoctoral researcher at KIT. He is an expert in deductive program verification and is particularly interested in applying its methods to the formal aspects of MDE. You can contact the author at ulbrich@kit.edu or visit <https://formal.kastel.kit.edu/ulbrich/>.

Anne Koziolok is a professor at KIT, Germany. She received her PhD degree from KIT in 2011. She is interested in MDE and agile development processes. You can contact the author at koziolok@kit.edu or visit https://mcse.kastel.kit.edu/staff_Koziolok_Anne.php.