JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

CoCoPath: Concolic Exploration of Consistency-Preserving Paths

Bowen Jiang*, **Miriam Boss***, **Thomas Weber***, **Weixing Zhang***, **Mattias Ulbrich***, **and Anne Koziolek***
*Karlsruhe Institute of Technology, Germany

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 Koziolek. *CoCoPath: Concolic Exploration of Consistency-Preserving Paths*. Journal of Object Technology. Vol. 25, No. 3, 2026.
http://dx.doi.org/10.5381/jot.2026.25.3.a9

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 models[1] 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 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] https://eclipse.dev/app4mc/help/latest/index.html#section3

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

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

Path In this context, an (execution-)path is an instance of an applied correct CPR per input combination, i.e., with the CPR definition there is exactly one execution path per input combination. With the integration of user input, an input combination may have multiple paths. From a more technical standpoint, an (execution-)path is a concrete control flow path through a transformation execution, determined by the outcomes of conditional statements that depend on user input. Additionally, path constraints can be used to impose further restrictions on behavior of a CPR on branching points. We call each conditional control-flow construct whose outcome depends on a CPR parameter a decision point. An execution path is then determined by the sequence of outcomes at all decision points encountered during a CPR execution.

2.2. Symbolic Execution and Concolic Execution

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

2.3. Dynamic Taint Tracking

Taint tracking originates from the security domain, where potentially malicious inputs are "tainted", i.e., tagged, to track where they are used, and whether any critical function might be affected. *Dynamic* taint tracking denotes that tags are propagated during program execution. It is also applicable for the general purpose of tracking flows through a program. For our prototype, we use the Reactions language of VITRUVIUS, which is compiled into Java bytecode; therefore, we use taint tracking for the Java Virtual Machine (JVM). There are multiple taint trackers available, e.g., Phosphor (Bell & Kaiser 2014), MirrorTaint (Ouyang et al. 2023), or Galette (Hough & Bell 2025). We use Galette because it is the most compatible with modern JVM versions, which are used in VITRUVIUS.

3. Related Work

Consistency Preservation Tools Several tools have been developed to manage consistency between models, e.g., DesignSpace (Demuth, Riedl-Ehrenleitner, et al. 2015), OpenFlexo (Golra et al. 2016), MetaEdit+ (Kelly et al. 1996; Kelly & Tolvanen 2021), Comprehensive Systems (Stünkel et al. 2021), or openCAESAR (Elaasar et al. 2023). While they share common goals with VITRUVIUS in general and also with CoCoPath and its goal of developer guidance, they differ in the way they treat user input and in parts the explicit definition of consistency rules. CoCoPath exploits both for concolic exploration of consistency-preserving paths under user-driven changes, providing guidance to developers in their task to preserve consistency in a multi-model environment.

Model Transformation Approaches For the broader field of model transformations, several approaches deal with related questions. Eramo et al. (2015) generate multiple consistent models from an underspecified relational transformation in JTL and represent them compactly using uncertainty models. In contrast, CoCoPath analyzes the execution paths of procedural consistency-preserving repair programs with symbolic inputs to systematically explore possible repair outcomes. Egyed et al. (2011) highlight the central role of human guidance in model transformation, arguing that change propagation across heterogeneous models often cannot be fully automated due to missing or ambiguous information in their vision of a smart transformation assitant. Kretschmer et al. (2021) systematically explore sequences of repairs to models that have been changed by developers, similarly to CoCoPath. In contrast to their approach, we exploit the, in our approach defined and used, consistency rules for computing such sequences. Constraint-driven Modeling (CDM) (Demuth, Lopez-Herrejon, & Egyed 2015) expresses the implications of changes to one model through constraints on another model. Laghouaouta & Laforcade (2020) investigate uncertainty within model transformations by introducing partial patterns, which allow a transformation rule to encode multiple possible matching and production scenarios. Famelis et al. (2013) employ *May Models* to explicitly model uncertainty in models and lift graph-rewrite transformations to operate correctly on those models. Outside model transformations, efficient exploration has also been studied in directed controller synthesis (Ubukata et al. 2025).

Symbolic Execution for Java **Java Pathfinder** (Păsăreanu & Rungta 2010) is a symbolic execution engine for Java bytecode that systematically explores program paths by combining 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 $c_p : M_1 \times M_1 \times M_2 \to (T \to M_2)$. 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: $c_{dp} : (m_1 : M_1) \times M_1 \times (m_2 : M_2) \to T(m_1, m_2) \to M_2$. 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

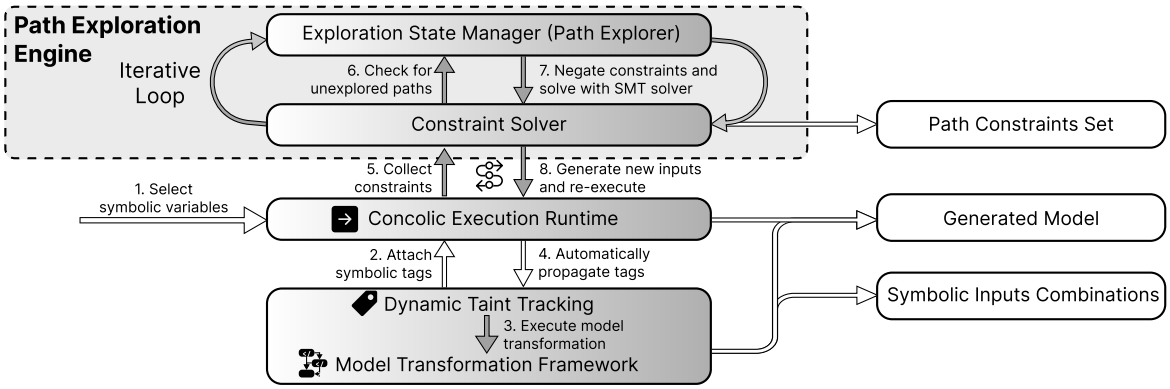

Figure 1 Overview of the CoCoPath approach

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

The aim of the CoCoPath approach is the following:

> For given $m_1, m_1' \in M_1$ and $m_2 \in M_2$, find a set of parameters $T_0 \subseteq T$ such that $\{i \mid 1 \leq i \leq k,\ c_p^{(i)}(m_1, m_1', m_2)(t),\ t \in T_0\}$ has maximum cardinality.

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

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

4.3. Running Example

We now present a small running example of BrakeSystem-ControlSystem to exemplify our approach for the rest of this section. The source metamodel (`BrakeSystem`) and the target metamodel (`ControlSystem`) represent two engineering views. `BrakeSystem` has a set of `BrakeDisc` model elements with a `diameter` attribute, while `ControlSystem` has a corresponding set of `AxleControlUnit` model elements. The consistency relation CR specifies that each `BrakeDisc` has a corresponding `AxleControlUnit` whose `controlProfile` records the assigned drive mode, and whose `absDecelThreshold` represents the target ABS deceleration limit for the axle unit and is computed as $diameter/50 \times multiplier$, where the multiplier is determined by a user input.

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

4.4. Symbolic Inputs and Symbolic Propagation

CoCoPath treats awaited user inputs as symbolic variables during execution. Conceptually, user inputs originate from interaction points in the consistency-preserving process and correspond to parameters $t \in T$ in the formalization of parameterized CPRs. For the purpose of systematic path exploration, CoCoPath represents these user inputs as symbolic inputs, which range over an admissible finite or bounded domain and are associated with concrete runtime values before execution.

To propagate symbolic identities through the execution, CoCoPath employs *dynamic taint tracking* on the JVM. In particular, Galette is used to attach and propagate tags alongside concrete values, without altering program semantics. Importantly, taint tracking in CoCoPath is used *only* to propagate symbolic identity (and derived symbolic expressions) through data flow; it does not by itself prescribe how path constraints are constructed.

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

4.5. Path Constraint Construction

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

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

modifications.

This mechanism requires that the bytecode instrumentation framework is compatible with the transformation runtime, including class loading and reflective execution patterns (e.g., Open Services Gateway initiative (OSGi)).

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

This mechanism trades full automation for framework compatibility while preserving systematic path exploration. In particular, it avoids dependence on bytecode-level branch interception and remains compatible with complex execution stacks. In future work, automation could be increased by weaving in the standardized helper calls automatically, for example, when transforming a VITRUVIUS Reaction to its corresponding code. Still, as a limitation, this approach cannot handle decision points in third-party libraries.

Relationship Between the Mechanisms Mechanism A and Mechanism B are complementary: both aim to construct logically equivalent path conditions for decisions that depend on symbolic inputs. The difference lies in how decision points are detected—either implicitly via bytecode-level branch interception (Mechanism A) or explicitly via CPR-level registration (Mechanism B). Which mechanism is employed is an implementation choice and may depend on the characteristics of the transformation framework and runtime environment (e.g., OSGi). This dual design allows CoCoPath to maximize automation where bytecode-level interception is feasible, while retaining applicability to complex MDE frameworks through CPR-level constraint registration.

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

In our running example (Table 1), the branch `if (pr < 0)` in $\varphi^{(1)}$ is the decision point based on the symbolic input pr. In Mechanism A, the interceptor would automatically detect this branch when the guard is evaluated against the tagged value and the path condition (PC) is recorded without any changes to the CPR. In Mechanism B, the CPR explicitly records the PC through the standardized helper at the branch point. In both cases, the PC is the same: $\mathrm{PC}_1 = \{pr < 0\}$ at the first execution with $pr = -1$, intersecting with T to obtain $[-1, 0)$, as shown in the PC and $\mathrm{PC} \cap T$ columns. In the evaluation presented in this paper, admissible input domains are derived directly from the user interaction mechanism, and no additional domain constraints are required.

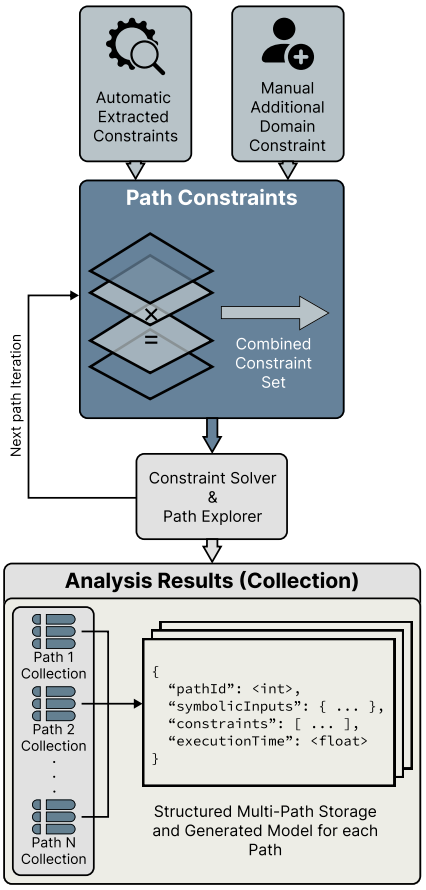

Figure 2 Systematic path exploration algorithm

4.6. Systematic Path Exploration Loop

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

In our running example, Table 1 shows the full exploration. Starting with $pr = -1$, $\varphi^{(1)}$ records $\mathrm{PC}_1 = \{pr < 0\}$ and the solver negates it to generate $pr = 0$. Successive executions cover $\varphi^{(2)}$–$\varphi^{(4)}$, with the solver generating $pr = 34$ and $pr = 67$ from the Solver-Generated Input column. After $\varphi^{(4)}$, there

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 >= 0 && pr < 34)`	$\{\ pr \geq 0,\ pr < 34\ \}$	$[\ 0,\ 34)$	$pr = 34$
$\varphi^{(3)}$	34	comfort	`if (pr >= 34 && pr < 67)`	$\{\ pr \geq 34,\ pr < 67\ \}$	$[34,\ 67)$	$pr = 67$
$\varphi^{(4)}$	67	sport	`else` ($pr >= 67$)	$\{\ pr \geq 67\ \}$	$[67,\ 100]$	—

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

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

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

5. Prototypical CoCoPath Implementation

This section describes the prototypical implementation of CoCoPath and its integration with the VITRUVIUS framework, illustrated in Figure 3. The prototype instantiates the methodology presented in Section 4, in particular the separation between symbolic propagation and constraint construction. The CoCoPath approach is language-agnostic: it applies to any CPR that can be represented as a program with branching control flow (Section 4.2), provided a suitable dynamic taint-tracking tool is available for the runtime. The current prototype is *Java-specific* because it relies on Galette (Section 5.2), which instruments JVM bytecode. In principle, other taint frameworks could serve the same role for different languages. Pure symbolic execution is a conceptual alternative, but is unlikely to scale to complex MDE frameworks such as VITRUVIUS (Section 6.7). Additionally, the current prototype targets CPRs implemented in *Java* within the VITRUVIUS framework.

The current prototype realizes *CPR-level constraint registration* (Mechanism B) together with optional domain constraints. Bytecode-level constraint extraction (Mechanism A) is not implemented in the prototype for VITRUVIUS. This is due to practical limitations: VITRUVIUS relies on a modern Java 17 runtime, extensive reflection, and OSGi-based class loading, which currently prevents reliable interception of all control-flow decisions at the bytecode level. We have validated the feasibility of bytecode-level constraint extraction (Mechanism A) for plain Java model transformations in a separate prototype based on Galette, which can intercept native bytecode comparisons [2]. This indicates that the limitation is not inherent to Java 17, but rather to the complexity of the VITRUVIUS execution stack. However, this prototype is not evaluated in this paper.

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

During execution, CoCoPath observes the concrete transformation runs, collects path constraints, and forwards them to the constraint solver. By negating previously observed path conditions, the constraint solver generates new concrete input assignments, which are reinjected to systematically explore alternative execution paths. For each execution, CoCoPath records the generated target model, the concrete input values, the collected path constraints, and execution time, providing analyzable feedback on alternative repair decisions.

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

5.1. Model Transformation Framework - VITRUVIUS

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

5.2. Dynamic Taint Tracking with Galette

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

5.3. CPR-Level Constraint Construction

In the current prototype, path constraints are constructed explicitly at the level of CPR implementations, corresponding to Mechanism B introduced in Section 4.5. Symbolic information is propagated automatically by Galette, while constraint construction is triggered only at explicit decision points in the transformation logic. Following Mechanism B (Section 4.5), constraint construction is triggered at explicit decision points in

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

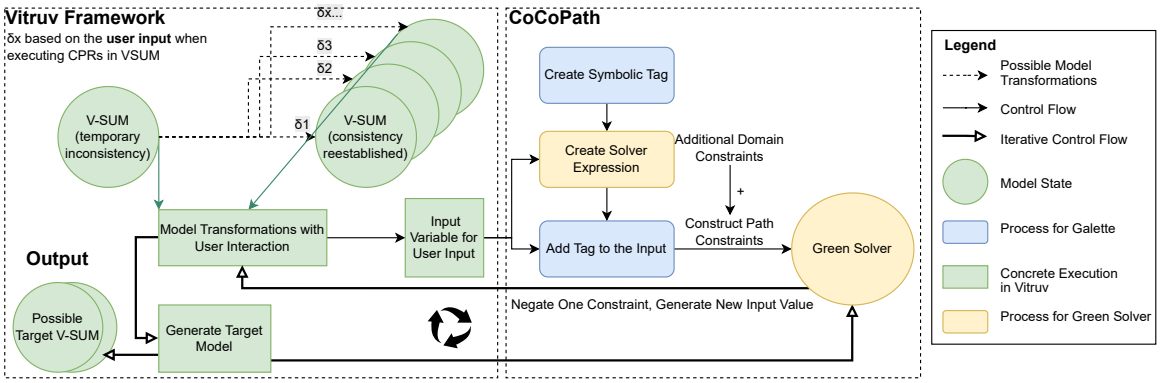

Figure 3 Prototypical CoCoPath implementation

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

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

5.4. Constraint Solving with the GreenSolver

The GreenSolver (Visser et al. 2012) is used in CoCoPath for transforming symbolic path and domain constraints into concrete input values for subsequent executions. CoCoPath represents both constraint types as symbolic expressions that are associated with tagged variables. The Solver collects these expressions from the constructed constraint sets, translates them into Green's internal representation, and delegates satisfiability checking to an SMT solver backend. In our current prototype, Green is configured to interface with the SMT solver Z3.

6. Evaluation

To address the RQs introduced in Section 1, we evaluate CoCoPath through two case studies: Consistency preservation between AMALTHEA and ASCET models based on an industrial case from Bosch and an extension of the running example.

We first describe the industrial context and the AMALTHEA–ASCET scenario that induces user-driven branching decisions (Section 6.1). We then explain how the scenario was integrated into our CoCoPath prototype (Section 6.3) and summarize the experimental setup used for all measurements (Section 6.4). We then present our results for RQ1 (Section 6.5), RQ2 (Section 6.6), RQ3 (Section 6.7), and RQ4 (Section 6.8). We discuss threats to validity in Section 6.9.

6.1. AMALTHEA-ASCET case study (A^2 case)

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

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

A^2 case We evaluate CoCoPath using an adapted consistency-preservation scenario from the automotive domain described by Mazkatli (2016). The evaluated CPRs are triggered when a new `Task` is added to an AMALTHEA model and require the creation of a corresponding task element in the ASCET model.

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

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

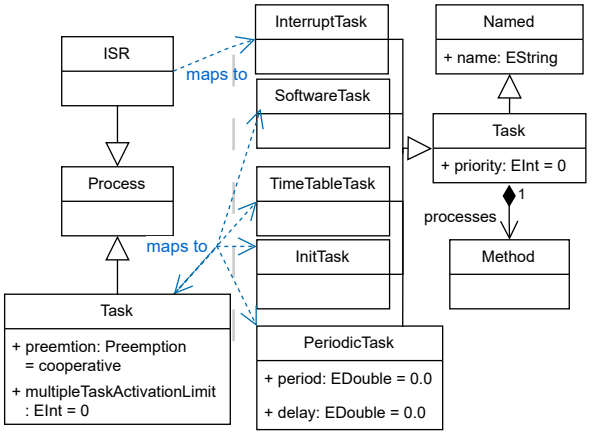

Figure 4 Snippet of the AMALTHEA metamodel (left), the ASCET metamodel (right), and the mapping relations (blue), based on a simplified scenario in Mazkatli (2016).

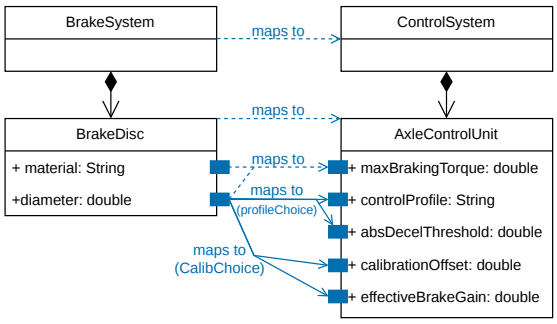

Figure 5 Snippet of the BrakeSystem metamodel (left), the ControlSystem metamodel (right), and the mapping relations (blue). Solid arrows: mappings requiring user input.

result, the impact of a decision is not obvious without executing the transformation, making this scenario well-suited to evaluate CoCoPath's systematic exploration of decision-dependent execution paths.

6.2. BrakeDisc-ControlSystem case study (BCCS case)

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

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× despite a 5× 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)

$\prod_{i=1}^{d} b_i$, corresponding to the classical path-explosion problem known from symbolic execution (Cadar & Sen 2013). For each path, CoCoPath performs one concrete execution of the CPR chain and one SMT solver query generated from the negated path condition. The solver cost depends on the theory of the path constraints. In the current prototype, constraints consist only of linear equalities and inequalities, which can be solved efficiently even for many variables. More expressive constraint theories such as bitvectors or strings are also supported by modern SMT solvers, although their solving complexity can be higher in the worst case (De Moura & Bjørner 2008). In the CoCoPath setting, however, path constraints typically involve only a small number of symbolic variables originating from user inputs, so we expect feasible solver cost even when additional theories are used. Furthermore, CoCoPath explores only feasible paths; infeasible path conditions are pruned automatically by the SMT solver. Since exploration starts from a concrete model state and a specific change and assuming VSUMs with low coupling and high cohesion, we expect the number of feasible paths to 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 scalability for multi-variable path exploration. Despite the framework initialization accounting for the majority of the total runtime for small workloads, the amortized per-path execution cost is relatively stable as the decision space increases, enabling 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 applicability 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 frameworks. VITRUVIUS relies on EMF-based models, extensive reflection, and dynamic class loading, and execution within an OSGi runtime environment. The results reveal that existing symbolic 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 incompatibilities, 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:

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

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

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

8. Conclusion

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

Acknowledgments

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

References

ASCET Repository. (2024). Retrieved 2025-12-12, from https://github.com/etas/ascet

Atkinson, C., Stoll, D., & Bostan, P. (2008). Orthographic software modeling: a practical approach to view-based development. In *International Conference on Evaluation of Novel Approaches to Software Engineering* (pp. 206–219).

Bell, J., & Kaiser, G. (2014). Phosphor: illuminating dynamic data flow in commodity JVMs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (p. 83–101). New York, NY, USA: ACM. doi: 10.1145/2660193.2660212

Brun, C., & Pierantonio, A. (2008). Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, *9*(2), 29–34.

Bruneliere, H., Burger, E., Cabot, J., & Wimmer, M. (2019). A feature-based survey of model view approaches. *Software & Systems Modeling*, *18*(3), 1931–1952.

Cadar, C., & Sen, K. (2013). Symbolic execution for software testing: three decades later. *C. of the ACM*, *56*(2), 82–90.

Cicchetti, A., Ciccozzi, F., & Pierantonio, A. (2019). Multi-view approaches for software and system modelling: a systematic literature review. *SoSyM*, *18*(6), 3207–3233.

De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 337–340).

Demuth, A., Lopez-Herrejon, R. E., & Egyed, A. (2015). Constraint-driven modeling through transformation. *Software & Systems Modeling*, *14*(2), 573–596.

Demuth, A., Riedl-Ehrenleitner, M., Nöhrer, A., Hehenberger, P., Zeman, K., & Egyed, A. (2015). DesignSpace: an infrastructure for multi-user/multi-tool engineering. In *Proceedings of the 30th Annual ACM SAC* (pp. 1486–1491).

Egyed, A., Demuth, A., Ghabi, A., Lopez-Herrejon, R., Mäder, P., Nöhrer, A., & Reder, A. (2011). Fine-tuning model transformation: Change propagation in context of consistency, completeness, and human guidance. In *Intl. Conf. on Theory and Practice of Model Transformations* (pp. 1–14).

Elaasar, M., Rouquette, N., Wagner, D., Oakes, B. J., Hamou-Lhadj, A., & Hamdaqa, M. (2023). openCAESAR: Balancing Agility and Rigor in Model-Based Systems Engineering. In *2023 ACM/IEEE MODELS-C*. doi: 10.1109/MODELS-C59198.2023.00051

Eramo, R., Pierantonio, A., & Rosa, G. (2015). Managing uncertainty in bidirectional model transformations. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering* (pp. 49–58).

Famelis, M., Salay, R., Di Sandro, A., & Chechik, M. (2013). Transformation of models containing uncertainty. In *Intl. conf. on Model-Driven Engineering Languages and Systems* (pp. 673–689).

Godefroid, P., Klarlund, N., & Sen, K. (2005, June). DART: directed automated random testing. *SIGPLAN Not.*, *40*(6), 213–223. Retrieved from https://doi.org/10.1145/1064978.1065036 doi: 10.1145/1064978.1065036

Golra, F. R., Beugnard, A., Dagnat, F., Guerin, S., & Guychard, C. (2016, March). Addressing Modularity for Heterogeneous Multi-Model Systems Using Model Federation. In *Companion Proceedings of the 15th International Conference on Modularity*. New York, NY, USA: ACM. doi: 10.1145/2892664.2892701

Hough, K., & Bell, J. (2025). Dynamic Taint Tracking for Modern Java Virtual Machines. *Proceedings of the ACM on Software Engineering*, *2*(FSE), 1757–1779.

Jiang, B. (2025a). *CoCoPath: Concolic Path Exploration for Consistency-Preserving Model Transformations*. https://github.com/IngridJiang/CocoPath.

Jiang, B. (2025b). *Implementation of the AMALTHEA-ASCET synchronization case study in VITRUVIUS*. https://github.com/IngridJiang/Amalthea-acset.

Jiang, B. (2026). *Implementation of the BrakeSystem-ControlSystem synchronization case study in VITRUVIUS*. https://github.com/IngridJiang/BrakeDiscwithUserinput.

Kelly, S., Lyytinen, K., & Rossi, M. (1996). Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In *International Conference on Advanced Information Systems Engineering* (pp. 1–21).

Kelly, S., & Tolvanen, J.-P. (2021). Collaborative modelling and metamodelling with MetaEdit+. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)* (pp. 27–34).

Klare, H., Kramer, M. E., Langhammer, M., Werle, D., Burger, E., & Reussner, R. (2021). Enabling consistency in view-

based system development - The Vitruvius approach. *Journal of Systems and Software*, *171*, 110815.
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.
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).
Laghouaouta, Y., & Laforcade, P. (2020). Dealing with uncertainty in model transformations. In *Proceedings of the 35th Annual ACM SAC* (pp. 1595–1603).
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.
Lucas, F. J., Molina, F., & Toval, A. (2009). A systematic review of UML model consistency management. *Information and Software technology*, *51*(12), 1631–1645.
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).
Mazkatli, M. (2016). *Consistency Preservation in the Development Process of Automotive Software* (Master's thesis). (ISBN: 9781543532913) doi: 10.5445/IR/1000143392
Mazkatli, M., Burger, E., Koziolek, A., & Reussner, R. H. (2017). Automotive systems modelling with Vitruvius. In *INFORMATIK 2017* (pp. 1487–1498).
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 *MODELSWARD* (pp. 117–128).
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
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).
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
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
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).
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
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).
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.

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 Koziolek 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 koziolek@kit.edu or visit https://mcse.kastel.kit.edu/staff_Koziolek_Anne.php.