[GRSS26] J. Grahl, B. Rumpe, M. Stachon, S. Stüber:
Dynamic Symbolic Execution of Component-and-Connector Models for Semantic Differencing.
In: Modellierung 2026, Volume P378, pp. 89-105, LNI, DOI 10.18420/modellierung2026-06, GI, Bayreuth, Mar. 2026.

Dynamic Symbolic Execution of Component-and-Connector Models for Semantic Differencing

Johanna Grahl[1], Bernhard Rumpe[1], Max Stachon[1] und Sebastian Stüber[1]

Abstract: In model-driven development, ensuring the correctness and consistency of evolving models is critical. This paper explores the use of Dynamic Symbolic Execution (DSE) for semantic difference analysis of Component and Connector (C&C) architectures, exemplified on MontiArc models. We extended the existing MontiArc-to-Java generator to capture both symbolic and concrete execution data at runtime, including transition conditions, visited states, and automata variables. This data enables the identification of key execution traces that reveal system behavior. We evaluate execution strategies based on runtime efficiency, minimality, and completeness. Finally, we asses DSE's applicability in semantic difference analysis. Our results show that DSE offers promise for analyzing component-and-connector architectures, though scalability remains a challenge.

Keywords: Dynamic Symbolic Execution, Model Analysis, Architecture Models, Component and Connector Architectures, MontiArc, Semantic Difference, Symbolic Code

1 Introduction

Model-driven approaches in software engineering enhance development by treating models as primary artifacts [FR07] and exploiting automation for tasks such as model-to-code translation [Ad20; HKR21; Ru17; SGT20]. Effective change management is crucial for maintaining model integrity during evolution. Semantic differencing operators support this process by automatically detecting semantic differences between successive model versions, ensuring that modifications do not introduce errors. These operators analyze the set of legal model instances defined by the model's denotational semantics [HR04], producing „diff-witnesses“ to illustrate the detected semantic differences.

While semantic differencing has been developed for various modeling languages [Dr19a; Dr19b; Ka21; MRR11a; MRR11c; RRS23; Ru24], the challenges of comparing dynamic and compositional Component and Connector (C&C) architectures remain underexplored. Existing methods often rely on translations to automata for comparison, which can only address a limited subset of C&C models [Bu17b; Bu19].

This paper is a revised and abbreviated version of an arXiv-preprint [Gr25]. In it, we propose a novel approach to semantic differencing for C&C architectures using Dynamic Symbolic Execution (DSE) [Ca11], which directly analyzes executable models without

[1] RWTH Aachen, Informatik 3 (Software Engineering), Ahornstraße 55, 52074 Aachen, Germany,
johanna.grahl@rwth-aachen.de; rumpe@se.rwth-aachen.de, https://orcid.org/0000-0002-2147-1966;
stachon@se-rwth.de, https://orcid.org/0000-0002-6328-3816;
stueber@se-rwth.de, https://orcid.org/0000-0002-6636-9375

requiring state-space translations. By applying DSE to MontiArc [Bu17a; Ha16; HRR12], an architecture description language for component-based systems, we can efficiently detect semantic differences in the behavior of models, including non-deterministic transitions and feedback loops.

We extend MontiArc's code generator to produce symbolic code for DSE and provide multiple execution strategies that employ the SMT solver Z3 [DB08]. Our approach is evaluated based on runtime, minimality, and completeness, and we discuss its limitations and potential optimizations.

The remainder of this paper is structured as follows: Section 2 provides the background on MontiArc, DSE, and semantic differencing, and discusses corresponding related work; Section 3 presents an example architecture; Section 4 describes the tool design; Section 5 evaluates the approach; Section 6 discusses results; and Section 7 concludes and provides an outlook on future work.

2 Background and Related Works

This section provides a brief overview of the theoretical and technical background on MontiArc, DSE, and semantic differencing, and discuss related works.

2.1 MontiArc

MontiArc [Bu17a; Ha16; HRR12] is a custom architectural description language (ADL) for modeling *component-and-connector* architectures. These kinds of architectures are used for the design of complex engineering or cyber-physical systems [Be17; Ku17; RW18]. The SysML standard [OM24] is, for example, largely based on component-and-connector architectures, and MontiArc itself is quite similar to SysML v2.

Components are the main structural elements and they may either be *atomic* or *composite* in nature, with the latter consisting of sub-architectures that enable hierarchical system design. Communication between components occurs through message-based channels connecting input and output ports. In both SysML and MontiArc, the input-output behavior of atomic components can be specified via state machines. MontiArc, in particular, uses a variant of Statecharts (SCs) [Ha87; Ru16] grounded in FOCUS semantics [BS12]. These automata allow *underspecification*, where inputs may lack defined transitions, and *non-determinism*, where multiple transitions exist for the same input. Both characteristics support flexible modeling of complex or partially defined systems.

MontiArc enables the modeling and simulation of executable architectures, making it suitable for exploring system behavior and for symbolic or dynamic analysis. Its modularity and formal foundations facilitate the integration of automated reasoning techniques — like symbolic execution – within the model-driven development process.

2.2 Dynamic Symbolic Execution

Dynamic Symbolic Execution (DSE) extends classical symbolic execution by combining *symbolic* and *concrete* program execution [Ca11]. In symbolic execution, program inputs are represented as symbolic variables, and each program path is characterized by a *path condition* – a conjunction of constraints on these inputs [Ki76]. Solving these constraints using an SMT solver produces concrete input values that drive specific program paths.

List. 1 shows a small example program with an unreachable branch. In symbolic execution, the resulting execution tree (cf. Figure 1) represents all explored paths, each labeled by its corresponding path condition (e.g., $x < 10$ and $x \geq 10$). Each leaf node denotes a satisfiable path condition.

```
1 public int example(int x){
2   int z = 2*x;
3   if(z < 10){
4     if(z > 10) { z = z + 1; } // unreachable
5     z = z * 2;
6   }
7   return z;
8 }
```

List. 1: Example of a program with an unsatisfiable path

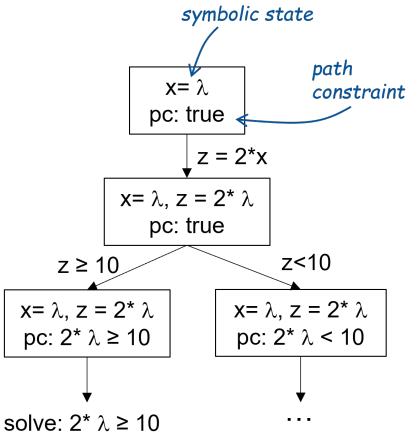

Figure 1: Symbolic execution tree. Arrows represent assignments or branches.

Pure symbolic execution is limited by path explosion and by constraints that are difficult to evaluate (e. g. non-linear arithmetic, cryptographic functions) [Go07]. DSE— also known as *concolic execution* — overcomes these issues by simultaneously executing concrete and symbolic values [Ba18; GKS05; SMA05; Xi13]. For example, executing the program with a concrete input $x = 6$ explores one specific path (cf. Figure 2). DSE then negates one branch

condition, uses an SMT solver to generate new input values (e. g. satisfying $x * 2 < 10$), and continues exploration.

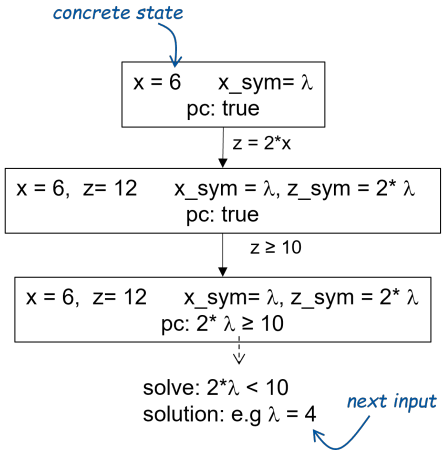

Figure 2: Dynamic Symbolic Execution with input $x = 6$.

Despite its power, DSE faces two key scalability challenges: *path explosion* and *constraint-solving complexity* [Ba18; GLM08; Wi21]. To address these, several tools integrate optimizations. CUTE [SMA05] applies bounded depth-first search and incremental constraint solving. JDart [Lu16; MH20], built on Java PathFinder [HP00; VPK04], analyzes Java programs through modular execution and search components. SAGE [GLM08] introduces *generational search* for efficient white-box fuzzing, while SMART [Go07] adopts *compositional analysis* to scale to larger systems.

Most existing DSE tools focus on code-level analysis. In contrast, our work applies DSE to executable *architectural models*, separating symbolic constraint collection from search strategy specification. This enables the semantic analysis of system models rather than program code, providing a foundation for semantic differencing.

2.3 Semantic Differencing

Semantic differencing is a form of comparative model analysis that examines differences in the legal instances of two models of the same modeling language according to their denotational semantics [HR04]. A key concept in semantic differencing is the notion of a *diff-witness*. If a legal instance exists in one model but not in the other, it is considered a diff-witness, indicating a semantic difference between the two models. Conversely, if no such instance exists, the former is a refinement of the latter, preserving its specifications and semantic properties. This technique is essential in ensuring the evolution of system models does not introduce unintended behaviors or errors.

Semantic differencing operators have been proposed for various modeling languages, including Activity Diagrams (ADs) [KR18; MRR11b; MRR11d], Class Diagrams (CDs) [MRR11c; RRS23], OCL [Ru24], Feature Diagrams (FDs) [Dr19b], Use Cases (UCs) [Ka21], and SCs [Dr19a] — including those used in C&C architectures [Bu17b; Bu19]. However, these approaches for SCs rely on translations to Büchi automata, which require finite state spaces and fixed input-output alphabets, making them unsuitable for more complex, infinite-state C&C architectures.

Our approach integrates DSE with semantic differencing to compare executable MontiArc models directly. By exploring and analyzing model behavior symbolically, it identifies semantic discrepancies without transforming the models into automata. This enables the comparison of C&C architectures that include non-determinism, internal variables, or feedback loops — scenarios that challenge traditional semantic differencing techniques.

3 Running Example

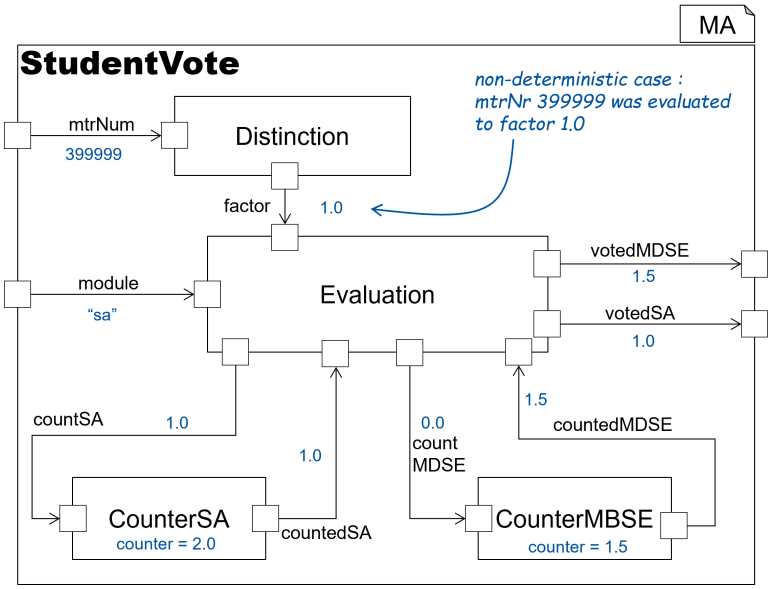

Figure 3: Architecture description of the `StudentVote`, with the representation of the internal state. The large, named boxes represent components, while the smaller adjacent boxes are ports. The arrows are connectors that represent the message flow between components.

In Figure 3, we illustrate the architecture of the MontiArc model `StudentVote`. The primary objective of this system is to survey university students and identify the most popular course from two available options: „Model-Based Systems Engineering“ (MBSE) and „Software

Architecture“ (SA). The `StudentVote` model is a composite structure that comprises four sub-components. The first sub-component, `Distinction`, calculates a weight for each student's vote based on their matriculation number. Older students, represented by lower matriculation numbers, are assumed to possess a more informed opinion; thus, their votes are assigned a weight of 1.5. Conversely, younger students, indicated by higher matriculation numbers, receive a weight of 1.0. The parameter defining this distinction within the `Distinction` component can induce non-deterministic behavior if set above 350000. Specifically, if a student's matriculation number falls between 350000 and the specified parameter, the weight of their vote can randomly be either of the two factors: 1.0 or 1.5.

The `Evaluation` sub-component is the second element in the `StudentVote` model. It receives the `factor` corresponding to the student's vote and forwards this value to the appropriate counter module. To cast a vote, the input must be either `"mbse"` or `"sa"`; any other input will result in the vote being disregarded. The two counters are interconnected within a feedback loop that includes the `Evaluation` component. To mitigate issues arising from the sequential processing of models, a delayed connection is implemented, allowing for effective feedback within the composite model. Each delayed port is assigned an initial value, and the resulting input value is stored for use in the subsequent processing step. Consequently, only multiple consecutive inputs can yield differing outputs. Notably, the first output generated by the `StudentVote` model is consistently 0.0, irrespective of the input provided.

In the following a possible execution, with input length 3 will be discussed. Initialization is done with parameter value 400000, creating a non-deterministic range between matriculation numbers 350000 and 400000. The first input is `(355555, "mbse")`, belonging to a student with matriculation number 355555 and vote for module MBSE. The delayed ports in the counter components, lead to output `(0.0, 0.0)`. The counter for module MBSE is displayed first, followed by the counter for module SA. However, the internal state of the model shows that a factor of 1.5 was assigned. Hence, `CounterMBSE` is incremented accordingly. The next student might generate the following input `(500000, "sa")`. The corresponding output `(1.5, 0.0)` represents the state after the first input. The internal state, however, differs, as the `CounterMBSE` is still 1.5 and `CounterSA` was incremented by 1.0. After the third input message, the output would be the state of the second input. A third input consisting of `(399999, "sa")` results in another non-deterministic case (see Figure 3). This time, the factor 1.0 was assigned, resulting in an increase of the `CounterSA` by 1.0. The output in step three `(1.5, 1.0)` shows the state after the second input.

The running example demonstrates how branches such as $mtrNum < 400000$ and non-deterministic behavior are handled. Additionally, the integer-values show that the approach works on large state-space.

4 Design and Concept

This section presents the key design decisions, a high-level overview of the tool's functionalities, and how we apply DSE to compute semantic differences between MontiArc models. The implementation is publicly available as part of the MontiArc project on GitHub[2]. For a more in-depth description refer to [Gr25].

4.1 Major Design Decisions

We chose DSE over traditional symbolic execution for two main reasons: it avoids infeasible paths that cannot be satisfied by any concrete values and it enables analysis of functions like cryptographic operations by using concrete inputs to generate symbolic information [Go07].

Rather than interpreting the models, MontiArc employs a code generator that produces executable Java code [Ha16]. We have extended this generator to allow for DSE by enabling the collection of symbolic and concrete information during execution.

Z3, an SMT solver by Microsoft, is used for predicate abstraction, test generation, and solving symbolic constraints [Bj19; DB08]. We interact with Z3 via the Java API, leveraging its support for multiple theories and model generation.

The tool currently supports automata with ports and internal variables of primitive types, strings, or enums, model composition, cyclic connector loops, parameters, and non-deterministic transitions.

4.2 Overview of the Tool's Functionalities

MontiArc models are converted into Java code using the extended generator. Execution collects information about states, transitions, and input-output behavior, which is transmitted to a controller that defines *interesting inputs* and manages execution strategies (see Figure 4).

Symbolic Java Code Variables are represented using the `AnnotatedValue` class, storing both symbolic and concrete values (List. 2). Transitions are generated as `if`-queries that communicate with the controller (List. 3).

Controller Functionality Controllers implement execution strategies and input generation. They collect branch conditions and state information to support path coverage, termination conditions, or random input generation. Multiple controller categories are supported:

[2] https://github.com/MontiCore/montiarc

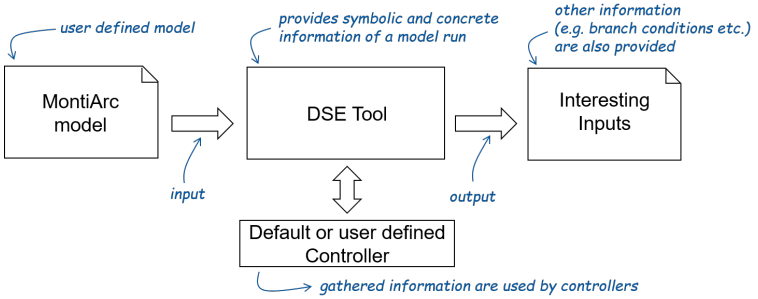

Figure 4: High-level overview of the developed tool

```
public class AnnotatedValue<SMTExpr extends Expr<? extends Sort>, T>{
  private final SMTExpr expr;  // From Z3 SMT Solver
  private final T value;       // Original Message
  //...
}
```

List. 2: AnnotatedValue class

```
BoolExpr expr = ctx.mkGT(matr.getExpr(), ctx.mkInt(35000));
if(TestController.getIf(expr, mtrNr.getValue() > 350000, "branchId")) {...}
```

List. 3: Generated if-query for a transition

- **Path Coverage:** Explore all model paths and implement oracles to handle non-determinism.
- **Termination Condition:** Stop based on specific termination conditions, e. g. visited states or transitions.
- **Random Generation:** Execute once or generate inputs randomly without symbolic reasoning.

Path Redundancy and Non-Determinism Redundant paths with identical simplified outputs can be merged to reduce analysis. Non-deterministic paths are detected by comparing symbolic conditions pairwise and checking satisfiability. While computationally expensive, this identifies multiple valid outputs for the same input.

Computation of Semantic Differences Between MontiArc Models Semantic differences are computed by executing one model using DSE, then applying the input-output pairs to a second model. If outputs differ after simplification, the input-output pair is a *diff-witness*. Non-determinism in the second model is handled by considering all oracle choices to ensure sound differencing.

5 Evaluation

In this section, we evaluate the tool and the implemented controllers using the `StudentVote` example introduced in Section 3 as our case study. We first define the evaluation criteria, followed by a presentation of the results, including the calculation of semantic differences between two MontiArc models.

5.1 Definition of Evaluation Criteria and Controllers

We established the following criteria for evaluating the implemented controllers: *Runtime*, *Minimality*, and *Completeness*.

Runtime The runtime of the developed tool is crucial in determining its usability. For comparison, each controller's runtime is classified into one of the following categories:

Constant Runtime: Runtime remains unchanged regardless of input length.

Linear Runtime: Runtime increases proportionally with input length.

Exponential Runtime: Runtime grows exponentially with input length.

Controllers with constant runtime are most desirable, followed by linear runtime, while exponential runtime is least favorable.

Minimalism The set of interesting inputs collected during analysis should be minimized to simplify outputs. Due to Z3's non-deterministic selection of concrete values, we focus only on symbolic values. Simplifying each symbolic output removes information about traversed paths, effectively treating the model as a black box.

For example, two interesting inputs may have symbolic outputs `x + 1 - 4` and `x - 5 + 2`, with branch conditions `x + 1 < 5` and `x - 5 < 10`. Both simplify to `x - 3`, and are considered duplicates under the *minimality* criterion.

Minimalism is measured as the proportion of duplicates relative to the size of the interesting input set. A higher percentage of duplicates is less favorable.

Completeness When using automata to describe component behavior, completeness can be assessed from multiple perspectives:

Visited Transitions: Ratio of visited transitions to total transitions.

Visited States: Ratio of visited states to total states.

States can be considered with or without including internal variable values. If the state space is infinite due to internal variables, absolute counts of visited states and transitions are used.

5.2 Results of the Evaluation

We evaluated the controllers using `StudentVote` as the case study. All calculations and runtime measurements correspond to the hardware used: a ThinkPad T14s with 32GB RAM, AMD Ryzen 7 Pro 2.7 GHz (8 cores).

Results are visualized in Figure 5, where red indicates a comparatively poor performance and green indicates a comparatively good performance. `Runtime` is marked as bad if it is exponential, while a constant runtime is considered good. For `Completeness`, a coverage of <25% of states and transitions is marked as poor and >75% is marked as good. For `Minimalism`, we measure the ratio inputs that produce a duplicate output compared to all inputs: a ratio of >75% is marked as poor, <25% is marked as good.

	Path Coverage				Termination Conditions					Random Generation	
	Path Coverage	PC Garbage Collector	Random Negation	PC Oracles	Termination Transitions	Termination State (excl. int. var.)	Termination State	Boring Interesting State (excl. int. var.)	Boring Interesting Transitions	Run Once	Random Inputs
Transition Completeness in %	100	100	100	100	46	92	92	100	92	38	46
States (incl. internal variables) Completeness in %	~100	~100	33	~100	10	31	31	42	40	6	6
States (excl. internal variables) Completeness in %	100	100	100	100	100	100	100	100	100	67	67
Duplicates	99	99	99	99	50	70	70	99	99	0	91
Speed	exponential	exponential	exponential	exponential	constant	linear	linear	exponential	exponential	constant	constant

Figure 5: Results regarding the `StudentVote` based on input length 4

Runtime `Path Coverage` controllers exhibit exponential runtime; e.g., input length 6 requires 1.9 days. Exponential growth is due to the number of solver calls.

`Termination Condition` controllers show variable runtime:

- Transition-based termination: constant runtime
- State-based termination: linear runtime

`Boring Interesting State/Transition` controllers, which rely on specific states or transitions for their termination conditions, may experience exponential runtime, depending on the chosen state or transition as well as the non-deterministic behavior of the SMT solver.

`Random Generation` controllers have constant runtime. `RunOnce` executes once per input length, `RandomInput` explores 10 paths.

Overall, `Random Generation` controllers are most efficient, followed by `Termination Condition`, while `Path Coverage` and `Boring Interesting` are least efficient.

Minimalism `StudentVote` allows multiple inputs to produce the same output. For input length 3, duplicates range from 86% to 92%, increasing to 91%-99% for length 4. Exceptions include `Termination Transition`, `Termination Automaton State`, `Termination State`, and `RunOnce`. This is due to the `RunOnce` controller only executing a single path, while the other exceptions can be attributed to their termination conditions restricting interesting inputs.

Completeness of Visited Transitions Full transition coverage requires all transitions to be visited. `Path Coverage` and `Termination Condition` controllers achieve 54% coverage (7 transitions) for input length 1, and 100% coverage (13 transitions) for input length 3. `Termination Transition` is limited by visit restrictions.

`Random Generation` controllers cover 31%-46% (4–6 transitions), independent of input length, due to limited iterations.

Completeness of Visited States Excluding Internal Variables `Path Coverage` and `Termination Condition` controllers visit 100% (6 states) from input length 1. `Random Generation` reaches only 67% (4 states).

Completeness of Visited States Including Internal Variables `StudentVote` has an infinite state space due to counters. Coverage depends on input length:

- `Path Coverage`: 100% (except `RandomNegation` at 33% for length 4)
- `Termination Condition`: declines with input length (10–40% for length 4)
- `Random Generation`: 6% for length 4

No single controller is universally superior; selection depends on the model and user objectives. For our purposes, we are interested in their suitability for semantic differencing, a quality we will be discussing in the following section.

Solver Optimization via Timeout Excessive solver calls (average CPU load 73%) motivate timeout implementation. Timeout reduces runtime but may degrade results. For the `Path Coverage` controller with garbage collection, an optimal timeout of 10 ms balances runtime improvement and minimal result degradation.

5.3 Computation of Semantic Differences Between MontiArc Models

To evaluate our semantic differencing approach, we compare our running example `StudentVote` (Figure 3) to a slightly different model `StudentVoteAlt`. In this alternative version of the model the `Evaluation` component now allows simultaneous voting for both modules through the message `mbse&sa`, resulting in each counter being incremented by 2.0. This addition creates a new execution path that is absent in the original `StudentVote` model.

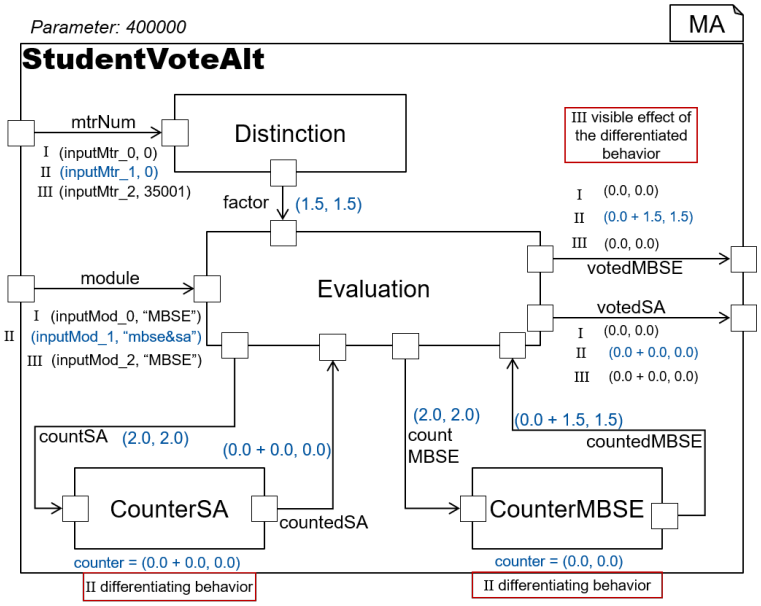

Figure 6: Semantic difference example between `StudentVoteAlt` and `StudentVote` for input length 3

Furthermore, each counter component is designed to reset to 0 once its internal value reaches 1.0. After this reset, the counter continues to count arbitrarily. Consequently, semantic differences are only observable when the input length exceeds 2. The relationship between input and output — specifically, the diff-witnesses between these two models — is illustrated in Figure 6.

Controller choice affects semantic difference calculation:

All diff-witnesses: `Path Coverage` recommended

Single diff-witness: Category 2 controllers sufficient

Using `Path Coverage Controller` with garbage collection, the semantic differences between `StudentVote` and `StudentVoteAlt` are shown in Table 1.

Diff-witnesses appear only for input length $\geq$ 3. Challenges in computing semantic

Input length	Time (min)	#Diff-witness	#Solver calls
1	0.08	0	140
2	1.34	0	2380
3	45.46	512	37196
4	22227.27	n/a	n/a

Table 1: Results of semantic differences between `StudentVote` and `StudentVoteAlt`

differences mirror those of the controller; larger models or longer input lengths require optimization strategies discussed in Section 6.

6 Discussion

As demonstrated in Section 5, the runtime of our tool is largely constrained by the CPU load associated with SMT solver operations. Specifically, the number of solver calls increases exponentially with both input length and model size, creating a significant scalability challenge.

To mitigate this, we implemented a timeout strategy for the SMT solver. If no model is found within a specified time frame, it is assumed that no solution exists. While this improves runtime, it may compromise result completeness. Selecting an appropriate timeout for each controller is therefore crucial. Parallelization of the analysis is another potential solution, though documentation and implementation for Z3 parallelism remain limited.

Beyond solver timeouts, further optimization requires reducing the number of solver calls through carefully designed controller strategies. In Section 5, we evaluated three categories of controllers, each offering distinct trade-offs between completeness and runtime. Comprehensive path coverage may necessitate exponential runtime, whereas accepting partial coverage allows for linear or constant runtime with respect to input length.

A promising approach is the combination of controllers. Random input generation provides minimal coverage with constant runtime, while path coverage controllers deliver exhaustive results at the cost of exponential runtime. By alternating between these strategies—using symbolic information collected from random inputs to guide targeted path exploration—we can improve path coverage within a manageable runtime budget, as illustrated in Figure 7. Alternatively we may employ knowledge about syntactic differences in the model to construct appropriate conditions for `Boring Interesting State/Transition` controllers, thus focusing only on specific paths through the model where we suspect a semantic difference to occur.

Additional avenues for optimization include decomposition analysis of models, fast unsatisfiability checks, common sub-constraint elimination, and incremental solving, as utilized by jCUTE [Go07]. Future work should also focus on extending support for additional data

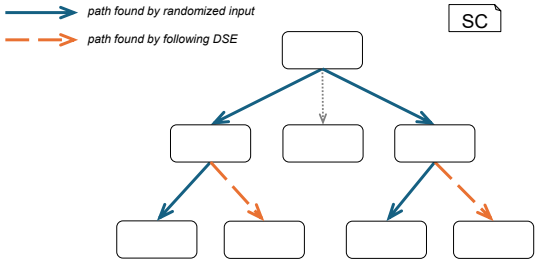

Figure 7: Possible paths found by a controller combining randomized inputs and DSE

types beyond primitives, strings, and enums, as well as investigating the development of a MontiArc interpreter to improve usability over the current generator-based approach.

7 Conclusion

In this paper, we presented a Dynamic Symbolic Execution (DSE) framework tailored for the MontiArc component-and-connector architecture language, implementing multiple execution strategies. Our approach enables semantic differencing between MontiArc models, effectively detecting behavioral differences while accommodating infinite state spaces and input-output alphabets.

We evaluated the framework and its controllers with respect to *runtime*, *minimality*, and *completeness*, identifying scalability as the primary limitation. While the approach is sound and capable of identifying diff-witnesses, runtime grows rapidly with input length, restricting applicability to larger models.

Building on the insights from Section 6, future work will explore optimization strategies such as parallelization, strategic early termination, and hybrid controller designs. We also plan to extend the tool to support additional data types and evaluate it against industrial and scientific MontiArc models. Beyond semantic differencing, Dynamic Symbolic Execution could further support automated test-case generation, input generation, and the identification of syntactic differences that lead to semantic changes. Finally, developing a MontiArc interpreter may enhance usability and broaden the scope of executable model analysis.

Together, these efforts aim to improve the scalability, efficiency, and applicability of DSE for component-and-connector architectures, ultimately providing a practical and versatile tool for model analysis and verification.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 250902306

References

[Ad20] Adam, K. et al.: Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In: 40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19). Bd. P-304. LNI, Gesellschaft für Informatik e.V., S. 59–66, 2020.

[Ba18] Baldoni, R. et al.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) 51 (3), S. 1–39, 2018.

[Be17] Bertram, V. et al.: Component and Connector Views in Practice: An Experience Report. In: Conference on Model Driven Engineering Languages and Systems (MODELS'17). IEEE, Austin, S. 167–177, 2017.

[Bj19] Bjørner, N. et al.: Programming Z3. Engineering Trustworthy Software Systems: 4th International School, SETSS 2018, Chongqing, China, April 7–12, 2018, Tutorial Lectures 4, S. 148–201, 2019.

[BS12] Broy, M.; Stølen, K.: Specification and development of interactive systems: focus on streams, interfaces, and refinement. Springer Science & Business Media, 2012.

[Bu17a] Butting, A. et al.: Architectural Programming with MontiArcAutomaton. In: In 12th International Conference on Software Engineering Advances (ICSEA 2017). IARIA XPS Press, Athens, Greece, S. 213–218, 2017.

[Bu17b] Butting, A. et al.: Semantic Differencing for Message-Driven Component & Connector Architectures. In: International Conference on Software Architecture (ICSA'17). IEEE, Gothenburg, S. 145–154, 2017.

[Bu19] Butting, A. et al.: Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution. Journal of Systems and Software (JSS) 149, hrsg. von Pelliccione, P.; Bosch, J.; Marija, M., S. 437–461, 2019.

[Ca11] Cadar, C. et al.: Symbolic execution for software testing in practice: preliminary assessment. In: 2011 33rd International Conference on Software Engineering (ICSE). S. 1066–1071, 2011.

[DB08] De Moura, L.; Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, S. 337–340, 2008.

[Dr19a] Drave, I. et al.: Semantic Differencing of Statecharts for Object-oriented Systems. In (Hammoudi, S.; Pires, L. F.; Selić, B., Hrsg.): Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19). SciTePress, Prague, S. 274–282, 2019.

[Dr19b] Drave, I. et al.: Semantic Evolution Analysis of Feature Models. In (Berger, T. et al., Hrsg.): International Systems and Software Product Line Conference (SPLC'19). ACM, Paris, S. 245–255, 2019.

[FR07] France, R.; Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. Future of Software Engineering (FOSE '07), S. 37–54, 2007.

[GKS05] Godefroid, P.; Klarlund, N.; Sen, K.: DART: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. S. 213–223, 2005.

[GLM08] Godefroid, P.; Levin, M. Y.; Molnar, D. A.: Automated Whitebox Fuzz Testing. In: Network and Distributed System Security Symposium. 2008.

[Go07] Godefroid, P.: Compositional dynamic test generation. In: ACM-SIGACT Symposium on Principles of Programming Languages. 2007.

[Gr25] Grahl, J. et al.: Dynamic Symbolic Execution for Semantic Difference Analysis of Component and Connector Architectures, arXiv, 2025, http://www.se-rwth.de/publications/Dynamic-Symbolic-Execution-for-Semantic-Difference-Analysis-of-Component-and-Connector-Architectures.pdf.

[Ha16] Haber, A.: MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems. Shaker Verlag, 2016.

[Ha87] Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming 8 (3), S. 231–274, 1987.

[HKR21] Hölldobler, K.; Kautz, O.; Rumpe, B.: MontiCore Language Workbench and Library Handbook: Edition 2021. Shaker Verlag, 2021.

[HP00] Havelund, K.; Pressburger, T.: Model checking java programs using java pathfinder. International Journal on Software Tools for Technology Transfer 2, S. 366–381, 2000.

[HR04] Harel, D.; Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"? IEEE Computer Journal 37 (10), S. 64–72, 2004.

[HRR12] Haber, A.; Ringert, J. O.; Rumpe, B.: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems, Technical Report AIB-2012-03, RWTH Aachen University, 2012.

[Ka21] Kautz, O.: Model Analyses Based on Semantic Differencing and Automatic Model Repair. Shaker Verlag, 2021.

[Ki76] King, J. C.: Symbolic Execution and Program Testing. Commun. ACM 19 (7), S. 385–394, 1976.

[KR18] Kautz, O.; Rumpe, B.: Semantic Differencing of Activity Diagrams by a Translation into Finite Automata. In: Proceedings of MODELS 2018. Workshop ME. Copenhagen, 2018.

[Ku17] Kusmenko, E. et al.: Modeling Architectures of Cyber-Physical Systems. In: European Conference on Modelling Foundations and Applications (ECMFA'17). LNCS 10376, Springer, Marburg, S. 34–50, 2017.

[Lu16] Luckow, K. et al.: JDart: A Dynamic Symbolic Analysis Framework. In (Chechik, M.; Raskin, J.-F., Hrsg.): Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Bd. 9636. Lecture Notes in Computer Science, Springer, S. 442–459, 2016.

[MH20] Mues, M.; Howar, F.: JDart: Dynamic Symbolic Execution for Java Bytecode (Competition Contribution). In (Biere, A.; Parker, D., Hrsg.): Tools and Algorithms for the Construction and Analysis of Systems. Springer International Publishing, Cham, S. 398–402, 2020.

[MRR11a] Maoz, S.; Ringert, J. O.; Rumpe, B.: ADDiff: Semantic Differencing for Activity Diagrams. In: Conference on Foundations of Software Engineering (ESEC/FSE '11). ACM, S. 179–189, 2011.

[MRR11b] Maoz, S.; Ringert, J. O.; Rumpe, B.: An Operational Semantics for Activity Diagrams using SMV, Technical Report AIB-2011-07, Aachen, Germany: RWTH Aachen University, 2011.

[MRR11c] Maoz, S.; Ringert, J. O.; Rumpe, B.: CDDiff: Semantic Differencing for Class Diagrams. In (Mezini, M., Hrsg.): ECOOP 2011 - Object-Oriented Programming. Springer Berlin Heidelberg, S. 230–254, 2011.

[MRR11d] Maoz, S.; Ringert, J. O.; Rumpe, B.: Modal Object Diagrams. In: Object-Oriented Programming Conference (ECOOP'11). LNCS 6813, Springer, S. 281–305, 2011.

[OM24] OMG: OMG Systems Modeling Language (SysML), Version 2.0 Beta 2, Beta 2 version of the SysML 2.0 specification, published April 2024, Object Management Group (OMG), 2024, https://www.omg.org/spec/SysML/2.0/Beta2/About-SysML.

[RRS23] Ringert, J. O.; Rumpe, B.; Stachon, M.: On Implementing Open World Semantic Differencing for Class Diagrams. Journal of Object Technology (JOT) 22 (2), 2:1–14, 2023.

[Ru16] Rumpe, B.: Modeling with UML: Language, Concepts, Methods. Springer International, 2016.

[Ru17] Rumpe, B.: Agile Modeling with UML: Code Generation, Testing, Refactoring. Springer International, 2017.

[Ru24] Rumpe, B. et al.: Semantic Difference Analysis with Invariant Tracing for Class Diagrams Extended by OCL. In: Workshop on Model Driven Engineering, Verification and Validation, MODELS Companion '24: International Conference on Model Driven Engineering Languages and Systems (MoDeVVa). Association for Computing Machinery (ACM), Linz, Austria, S. 1066–1075, 2024.

[RW18] Rumpe, B.; Wortmann, A.: Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In (Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, Hrsg.): Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday. LNCS 10760, Springer, S. 383–406, 2018.

[SGT20] Sebastián, G.; Gallud, J. A.; Tesoriero, R.: Code generation using model driven architecture: A systematic mapping study. Journal of Computer Languages 56, S. 100935, 2020, https://www.sciencedirect.com/science/article/pii/S2590118419300607.

[SMA05] Sen, K.; Marinov, D.; Agha, G.: CUTE: A concolic unit testing engine for C. In. Bd. 30, S. 263–272, 2005.

[VPK04] Visser, W.; Păsăreanu, C. S.; Khurshid, S.: Test input generation with Java PathFinder. In: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. S. 97–107, 2004.

[Wi21] Williams, N.: Towards exhaustive branch coverage with PathCrawler. CoRR abs/2105.05517, 2021, arXiv: 2105.05517.

[Xi13] Xiao, X. et al.: Characteristic studies of loop problems for structural test generation via symbolic execution. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). S. 246–256, 2013.