[GRSS25] J. Grahl, B. Rumpe, M. Stachon, S. Stüber:
Dynamic Symbolic Execution for Semantic Difference Analysis of Component and Connector Architectures.
DOI 10.48550/arXiv.2508.00749, arXiv, Aug. 2025.

Dynamic Symbolic Execution for Semantic Difference Analysis of Component and Connector Architectures

Johanna Grahl[1,†], Bernhard Rumpe[2,†], Max Stachon[2,†] and Sebastian Stüber[2,†]

[1]RWTH Aachen University, Germany
[2]Software Engineering, RWTH Aachen University, Germany

Abstract

In the context of model-driven development, ensuring the correctness and consistency of evolving models is paramount. This paper investigates the application of Dynamic Symbolic Execution (DSE) for semantic difference analysis of component-and-connector architectures, specifically utilizing MontiArc models. We have enhanced the existing MontiArc-to-Java generator to gather both symbolic and concrete execution data at runtime, encompassing transition conditions, visited states, and internal variables of automata. This data facilitates the identification of significant execution traces that provide critical insights into system behavior. We evaluate various execution strategies based on the criteria of runtime efficiency, minimality, and completeness, establishing a framework for assessing the applicability of DSE in semantic difference analysis. Our findings indicate that while DSE shows promise for analyzing component and connector architectures, scalability remains a primary limitation, suggesting further research is needed to enhance its practical utility in larger systems.

Keywords

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

1. Introduction

In model-driven software and systems engineering, models serve as the primary artifacts for development, evolving throughout their lifecycle due to modifications, refinements, and refactorings. To facilitate a less error-prone development process and ensure the preservation of critical model properties, automated model analyses, such as semantic differencing, can be employed [1].

Semantic differencing is a comparative model analysis technique that evaluates two models written in the same language by examining their legal instances as defined by a language-specific formal semantics [2]. A semantic difference is identified when a ""diff-witness" exists—an instance that is valid in one model but not in the other. Conversely, if no such witness is found, the first model can be considered a semantic refinement of the second. Various semantic difference operators have been developed for different modeling constructs, including activity diagrams [3, 4], class diagrams [5, 6], feature models [7], variants of statecharts [8, 9, 10], and sequence diagrams [11].

Despite the extensive exploration of semantic differencing in static structural models, such as class diagrams, and isolated behavioral models, like statecharts, component-and-connector architectures present unique challenges due to their dynamic nature and compositional complexity. Unlike static object structures that do not define dynamic behavior, component-and-connector models necessitate the consideration of both individual component behaviors and their interactions within compositions. This paper addresses these challenges by focusing on semantic differencing specifically for MontiArc models [12, 13, 14], a topic that has yet to be comprehensively covered in existing literature.

Understanding changes in architectural models is essential for early bug detection and ensuring correct refinements from underspecified designs. Semantic differencing is particularly effective in addressing

arxiv.com

† These authors contributed equally.

johanna.grahl@rwth-aachen.de (J. Grahl); rumpe@se.rwth-aachen.de (B. Rumpe); stachon@se-rwth.de (M. Stachon); stueber@se-rwth.de (S. Stüber)
https://se-rwth.de (B. Rumpe)
0000-0002-2147-1966 (B. Rumpe); 0000-0002-6328-3816 (M. Stachon); 0000-0002-6636-9375 (S. Stüber)

underspecification and refinement during the early stages of development, when architectural designs are abstract and not fully specified. As the design evolves, this initial underspecification is progressively refined through decomposition, architectural refactorings, and restrictions on component behavior. Our approach not only identifies semantic differences but also generates test cases that assist in subsequent development phases. These test cases provide valuable insights into specified behaviors and highlight areas that may require further specification or testing, thereby supporting a more robust model evolution process.

To analyze behavioral differences in component and connector architectures, we aim to leverage *Dynamic Symbolic Execution (DSE)* [15], a variant of symbolic execution [16], to develop a semantic differencing operator specifically for MontiArc models. Building upon the fundamental semantic-difference framework established in previous publications, we apply DSE as a novel technique for identifying semantic differences in these models.

MontiArc [12, 13, 14] is an architecture modeling framework developed with the MontiCore[1] language workbench [17, 18] that offers an architecture description language for designing both software and cyber-physical systems [19]. It enables modular hierarchical modeling [20] of architectural components and their message-oriented, asynchronous communication. The input-output behavior of atomic components can then be specified by a variant of Statecharts (SCs) [21, 22] based on the FOCUS formalism [23].

To facilitate simulations, MontiArc includes a code generator that produces executable Java code [13, 12]. We have extended this generator to create symbolic code suitable for DSE. This symbolic code can be invoked using a newly developed tool designed to initiate the analysis. A default execution strategy for DSE is provided, with support for additional user-defined strategies. These strategies utilize execution data from the MontiArc model and employ Z3 [24] as a satisfiability modulo theories (SMT) solver.

The information collected during execution includes both symbolic and concrete values, which serve as inputs for execution strategies and contribute to model validation efforts. An exemplary use case for this model validation is the calculation of semantic differences between MontiArc models. Specifically, a semantic difference is identified by a "diff-witness", which manifests as an input-output trace from the first model that cannot be replicated in the second model.

Our implementation supports primitive data types, strings, and enumeration types for ports and internal variables. Furthermore, it accommodates model composition, non-deterministic transition selection, and model parameters. Notably, we currently focus on component behavior specifications defined through SCs and time-synchronous communication between components.

Contributions In this paper, we present the following contributions:

- Development and implementation of DSE for MontiArc.
- Implementation of multiple DSE controllers, each employing distinct execution strategies.
- Realization of a DSE-based semantic differencing operator specifically for MontiArc models.
- Evaluation of the implemented controllers based on the criteria of *runtime*, *minimality*, and *completeness*, as well as their applicability to semantic differencing.
- Discussion on the utility and limitations of DSE in the context of component-and-connector architectures.

Our evaluation emphasizes completeness and runtime efficiency. However, it is important to note that enforcing completeness can lead to exponential increases in runtime with respect to input complexity. To address this limitation, we discuss potential mitigation strategies that can help balance these factors.

The remainder of this paper is structured as follows: In Section 2, we outline the concept of DSE. This is followed by an exploration of related work in the field of DSE in Section 3. Section 4 presents an example architecture to illustrate and evaluate our approach in the subsequent sections. The concept

[1] https://monticore.github.io/monticore/

and design decisions behind our tool's development are detailed in Section 5. In Section 6, we evaluate our DSE approach and the implemented execution strategies, followed by a discussion of the results and their implications in Section 7. Finally, we conclude in Section 8 with a summary of our findings and an outlook on future work.

2. Dynamic Symbolic Execution

Dynamic Symbolic Execution (DSE) is a variant of symbolic execution that generates concrete inputs alongside symbolic inputs [15]. In symbolic execution, each input is represented as a symbolic constant. Each execution path of a program is associated with a boolean expression known as the path condition. These path conditions are initialized with the value *true* and define the properties that must be satisfied by a concrete value in order to execute that specific path. During the process of symbolic execution, an *execution tree* is constructed, representing all paths identified by the chosen execution strategy [16]. Concrete values can be derived as inputs by solving individual path constraints using a Satisfiability Modulo Theories (SMT) solver.

Consider the program illustrated in listing 1, which takes an integer input x. In symbolic execution, an execution tree is constructed, and a symbolic constant (*e.g.,* λ) is applied to x. The path constraint is initialized with *true*, as depicted in fig. 1. While executing the program, assignments are also executed symbolically; for example, the assignment $z = 2 * x$ (cf. line 2) would be represented as $z = \lambda * 2$. At branching points, such as an `if` statement, the corresponding constraints are collected. The `if` condition in line 3 results in two possible paths, each with the following path constraints: $x < 10$x or $x \geq 10$. If the condition $x < 10$ is not satisfied, the execution of that particular path in the execution tree will terminate. Conversely, the alternative path is analyzed to completion. Each leaf node of the execution tree corresponds to a path constraint, representing the specific condition that must be satisfied for that execution path to be taken.

```
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 }
```

Listing 1: Example of a program with an unsatisfiable path

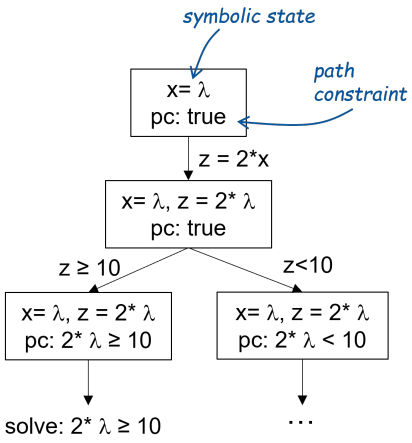

Figure 1: Symbolic execution tree. At the top is the initial state, arrows are assignments or branches.

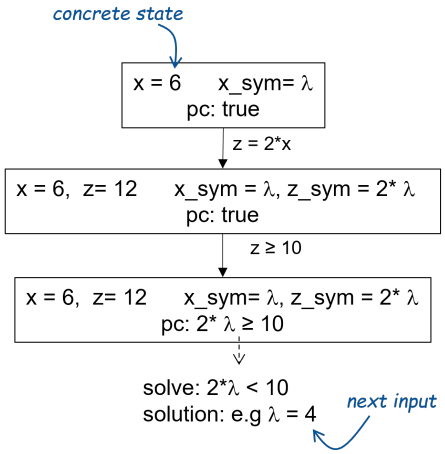

Figure 2: Representation of a specific path executed through Dynamic Symbolic Execution with input $x = 6$

Symbolic execution has inherent limitations, particularly in cases where certain functions cannot be effectively executed symbolically, such as cryptographic functions [25]. Dynamic Symbolic Execution (DSE), also known as concolic execution, addresses these limitations by combining both concrete and symbolic execution of a program, executing it with both concrete and symbolic values simultaneously [26, 27, 28, 29].

Referring back to our example program, we can arbitrarily choose a concrete value for the input, such as $x = 6$. This choice leads to a specific execution path, as illustrated in fig. 2 on the right side. Given that the concrete value for the variable z is calculated to be 12, the condition `z < 10` is not satisfied, resulting in the termination of this particular execution path.

When comparing the symbolic path information gathered from both symbolic execution and DSE, we find that there are no significant differences in the information collected. To explore new paths using DSE, the current path constraint is negated, and this modified constraint is then passed to an SMT solver. In our example, the corresponding formula would be $x * 2 < 10$. If this formula is satisfiable, the solver will produce a model containing concrete values that satisfy the condition. In this case, the solver might yield $x = 4$.

Further details regarding the design decisions related to symbolic execution and DSE for MontiArc are elaborated in section 5.1.

3. Related Work

Symbolic execution is a powerful technique used in program analysis that systematically explores program paths by treating inputs as symbolic variables. However, it is computationally intensive, and while advancements in modern computer hardware have facilitated the development of efficient theorem provers and scalable analysis tools, the challenges inherent in symbolic execution have prompted renewed interest in this area [25].

Despite its strengths, symbolic execution faces two primary limitations: path explosion and the complexity of constraint solving. These challenges are also prevalent in Dynamic Symbolic Execution (DSE) and significantly hinder scalability. For instance, constructs such as loops can lead to an exponential increase in the number of feasible execution paths, making it infeasible to systematically analyze all possible paths in large systems or programs. Additionally, the second limitation pertains to constraint solving, which is an NP-hard problem. In larger systems, path constraints can involve intricate combinations of conditions, further complicating the analysis. Consequently, achieving complete path coverage remains elusive in practice [28, 30, 29, 31].

To mitigate these limitations, several approaches have been proposed and integrated into various

tools. Notable examples include Microsoft's SAGE, CUTE, DART, and SMART, each designed to enhance the efficiency and effectiveness of symbolic execution in addressing the challenges of path explosion and constraint complexity.

CUTE [27], a concurrent unit testing engine, can efficiently explore paths in C code, achieving high branch coverage and bug detection. Due to the limitations of DSE with respect to path explosion, an execution strategy has been used to counteract this. CUTE uses a bounded depth-first search to counteract an infinite exhaustive search of the entire computation tree. Three optimizations have been implemented to counteract the limitations of constraint solving: first, the *fast unsatisfiability check*, which checks whether the last constraint of the path condition is syntactically the negation of any previous constraints, second, *common sub-constraints elimination*, in which the solver identifies and eliminates common arithmetic sub-constraints, and third, *incremental solving*, where the solver identifies dependencies between sub-constraints and exploits them to solve constraints faster and keep solutions similar. These optimizations reduce the number of sub-constraints and thus optimize the runtime.

jCUTE [32] was one of the pioneering symbolic execution tools designed specifically for Java programs. However, as jCUTE is no longer actively maintained, JDart has emerged as a robust replacement [33]. JDart employs DSE to analyze Java programs, primarily focusing on assertion checking. Through this approach, JDart can either identify assertion violations, exhaustively explore all program paths, or reach resource limits during analysis.

One of the key advantages of JDart is its capability to handle complex software systems, as exemplified by its development to analyze intricate NASA software. JDart's architecture is modular, comprising two primary components: the *executer* and the *explorer*. The *executer* is responsible for executing the program and collecting symbolic constraints, leveraging the Java PathFinder framework [34, 35]. In contrast, the *explorer* determines the search and execution strategy employed during analysis.

JDart was designed with two main objectives in mind. The first is to create a robust framework capable of managing industrial software challenges, such as handling crashes and addressing the constraints of constraint solving. The second objective is to establish a modular and extensible platform, allowing for the interchangeability of components, the use of various constraint solvers, and the implementation of multiple search strategies or termination conditions [36].

SAGE [30], developed by Microsoft, is an automated white-box fuzz testing tool widely utilized for detecting bugs and vulnerabilities in applications, such as those running on Windows. SAGE implements a technique known as *Generational Search*, which minimizes redundancy while maximizing the generation of new test cases. This systematic approach allows for effective analysis of the state space, enabling SAGE to manage large applications with substantial input sizes. Additionally, SAGE employs heuristics to enhance code coverage, further improving its effectiveness in identifying potential issues.

Another tool developed by Godefroid is DART [28], which focuses on automatic software testing using DSE. Building on DART's capabilities, SMART [25] introduces the concept of compositional analysis, wherein composite functions are decomposed into atomic functions for individual analysis. The results of these analyses are then synthesized through the use of pre- and post-conditions.

While many of the tools discussed earlier primarily focus on program analysis, our research is concerned with a different form of semantic model analysis. Notably, these tools typically employ interpretation rather than generating symbolic code. In our case, we opted to extend the existing code generator for MontiArc rather than develop a new interpreter from the ground up. Similar to JDart, our approach incorporates a modular architecture, separating the execution and collection of symbolic constraints from the implementation of search and execution strategies. Since our implementation is in its initial version, we have yet to incorporate optimizations to mitigate the challenges of path explosion and constraint solving; these enhancements are planned for future work.

A key aspect that differentiates our research from previous work on DSE is our intended goal: to ascertain the semantic differences between executable component-and-connector architecture models. As noted in section 1, semantic differencing operators have been developed for various modeling languages [37, 5, 7, 10, 11], and component-and-connector structures, such as statechart systems (SCS)

or automata, are no exception [8, 9]. However, these approaches typically rely on translations to Büchi automata, which necessitate a finite state space and a defined input-output alphabet. Additionally, to compare entire architectures, syntactic composition of the automata is required, posing challenges when feedback loops are present. Our DSE-based approach, in contrast, circumvents these limitations.

4. Running Example

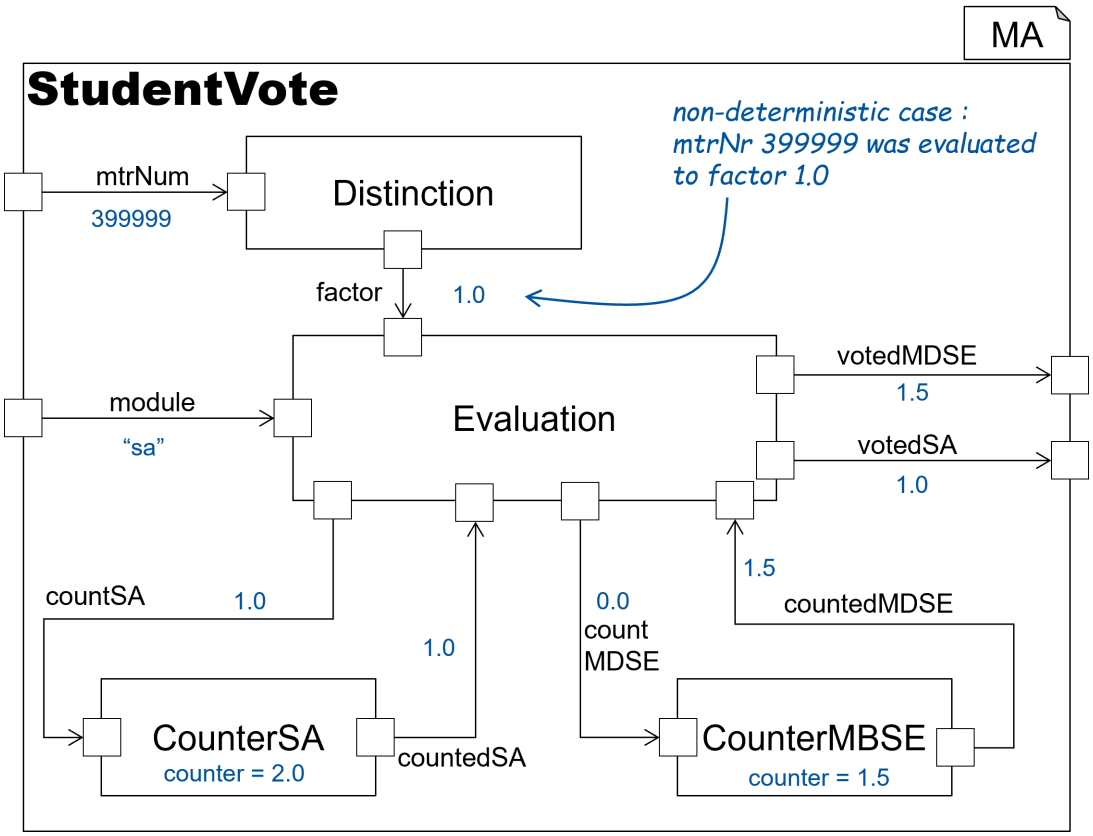

Figure 3: Architecture description of the `StudentVote`, with the representation of the internal state

MontiArc [12, 13, 14] is an architectural description language specifically designed for component-and-connector architectures. In this framework, components can either be atomic or consist of sub-architectures, allowing for a hierarchical organization of system elements. Communication between these components is achieved through a message-oriented approach, utilizing channels that connect output and input ports. The input-output behavior of atomic components is described by automata that are a variant of SCs [21, 22] based on the FOCUS formalism [23].

These automata facilitate underspecification due to partiality—where certain inputs may have missing transitions—and non-determinism, characterized by the existence of multiple transitions for the same input at a given state. In the following sections, we introduce a MontiArc model that will serve as the running example throughout this paper.

In fig. 3, we illustrate the architecture of the MontiArc model named `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 1.01.0. 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 0.00.0, 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. 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.

5. Design and Concept

This section discusses the main design decisions and provides a high-level overview of the developed functionalities and the implemented controller. Finally, we apply our Dynamic Symbolic Execution (DSE) approach to compute the semantic differences between two MontiArc models.

The implementation of the tool is publicly available as part of the MontiArc project on GitHub[2]

5.1. Major Design Decisions

This subsection outlines the key design decisions made during the development of the tool.

Dynamic Symbolic Execution vs. Symbolic Execution As discussed in section 1, DSE is a specialized variant of symbolic execution. We opted for DSE over traditional symbolic execution for two primary reasons.

First, when using symbolic execution, a program's analysis generates a tree structure containing all possible path combinations. This may lead to paths that cannot be satisfied by any concrete values. For instance, consider a function with nested `if` statements that have mutually exclusive conditions. Take, for example, the conditions `z < 10` and `z > 10`. In this case, one of the possible paths would require satisfying both conditions simultaneously, which is inherently impossible.

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

In contrast, DSE avoids constructing such infeasible paths, as it inherently recognizes that no concrete values can satisfy the path conditions.

The second reason for choosing DSE is that not all functions can be effectively analyzed using traditional symbolic execution. For instance, cryptographic functions, such as hash functions, cannot be evaluated through symbolic execution without undermining their security properties. In the case of DSE, these functions can be analyzed by utilizing concrete values for variables. By substituting concrete values, we can evaluate the function and consequently derive new input values for further analysis [25]. In summary, DSE was selected because it enables the analysis of functions that are not amenable to traditional symbolic execution and avoids exploring paths with unsatisfiable conditions.

Z3 as an SMT Solver Z3 is an award-winning SMT solver developed by Microsoft. It is primarily used for predicate abstraction, advanced static checking, and test case generation. Interaction with Z3 can occur through the SMT-LIB format as well as various APIs [24]. Our implementation utilizes the Java API for communication with the solver. Z3 supports multiple theories, including equality, arithmetic operations, and uninterpreted functions. Additionally, it generates a model containing concrete values for all defined constants within the formula. Z3 also includes numerous optimizations and tactics to enhance its performance [38].

Supported MontiArc Features In our approach, we currently limit the behavior descriptions of MontiArc models to automata that incorporate the following features: Ports and internal variables of the automata can be of primitive types, strings, or enumeration types. Additionally, the composition of models, cyclic connector loops, and the definition of model parameters are supported. Notably, our developed extension of the generator also facilitates non-deterministic transition selection, which was previously unsupported.

Generation instead of Interpretation Unlike other DSE engines that perform Dynamic Symbolic Execution by executing the program under test and collecting symbolic information without explicit code generation [30, 33, 27], MontiArc employs a code generator that produces executable Java code [13]. In this paper, we evaluate a DSE approach based on this existing code generator and extend it to generate symbolic code. We also discuss alternative methods, such as using an interpreter, in section 7.

5.2. Overview of the Tool's Functionalities

The input of the generator is defined as a MontiArc model. This model may only contain the supported MontiArc features, listed in section 5.1. MontiArc models are converted into symbolic Java code using the aforementioned extended generator. During the execution of the generated symbolic code, a variety of information is collected, including transition condition, taken transitions and state information. State information of an automaton can be considered with or without the symbolic and concrete values of internal variables. Collected information is transmitted to a controller, which defines the overall output of the tool and *interesting inputs*. Interesting inputs are input-output pairs and their corresponding branch conditions. Branch conditions represent the specific path taken, as they are the transition conditions satisfied by the corresponding input. A high level overview of the tool is displayed in fig. 4. In the following the main functionalities and features are presented.

Symbolic Java Code Collecting and analyzing information requires symbolic Java code. This code is generated by the extended code generator, with a significant enhancement being the ability to process both symbolic and concrete values of variables simultaneously. To achieve this, ports and internal variables are represented using a new data type called `AnnotatedValue` (see listing 2). The `AnnotatedValue` class features two attributes that represent the variable's symbolic and concrete values.

To extract the information encoded within `AnnotatedValue`, a second modification was made to the generator. In previous implementations, transitions were converted into if-queries, where the condition

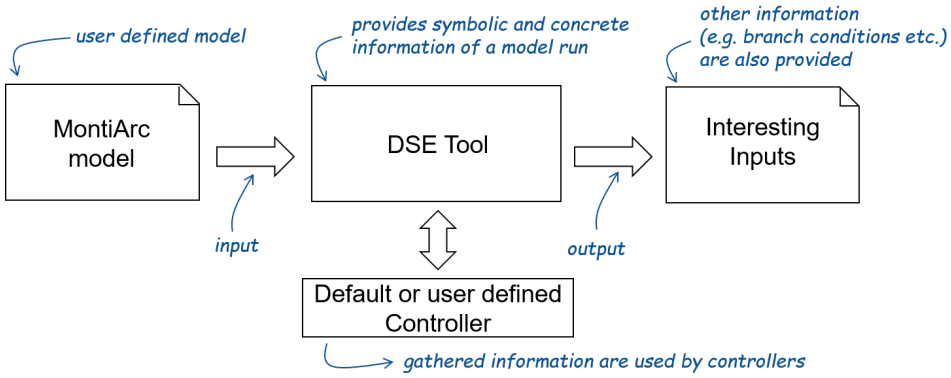

Figure 4: High level overview of the developed tool

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

Listing 2: Java class that stores symbolic expression in addition to value

```
BoolExpr expr = ctx.mkGT(matr.getExpr(),
                         ctx.mkInt(35000);

if(TestController.getIf(expr,
          mtrNr.getValue() > 350000, "branchId")
   ){...}
//...
}
```

Listing 3: Exampel of a generated if-query for a transition

represented the transition constraint. To systematically capture transition information, transitions are now generated as follows: for each transition, the function `TestController.getIf()` is invoked. This function takes the concrete values of all required variables, a boolean expression representing the symbolic condition, and the transition name as inputs. The controller then determines whether the transition can be executed and stores all relevant transition information. An example of such a generated if statement for a transition related to the matriculation number value is shown in listing 3.

Additionally, information about the current state of the model is collected at the end of each execution and passed to the controller for further processing.

In fig. 5, the model `StudentVote` from our motivating example is presented, with its internal state depicted through both symbolic and concrete values. A student with a matriculation number of 500000 votes for the module SA. Each internal variable and port is assigned a symbolic and concrete representation; for example, the input is represented as `(inputMtr_1, 500000)` and `(inputMtr_1, "sa")`.

For ports, only the current values are visible, as previous values are not retained. In contrast, internal variables, such as counters, may be influenced by prior inputs. The symbolic values of internal variables indicate how the current values were derived.

The expression `(0.0 + 0.0 + 1.0, 1.0)` represents the current symbolic state of the counter for SA. The initial and first input was 0.00.0, and the current computation has incremented the counter by one.

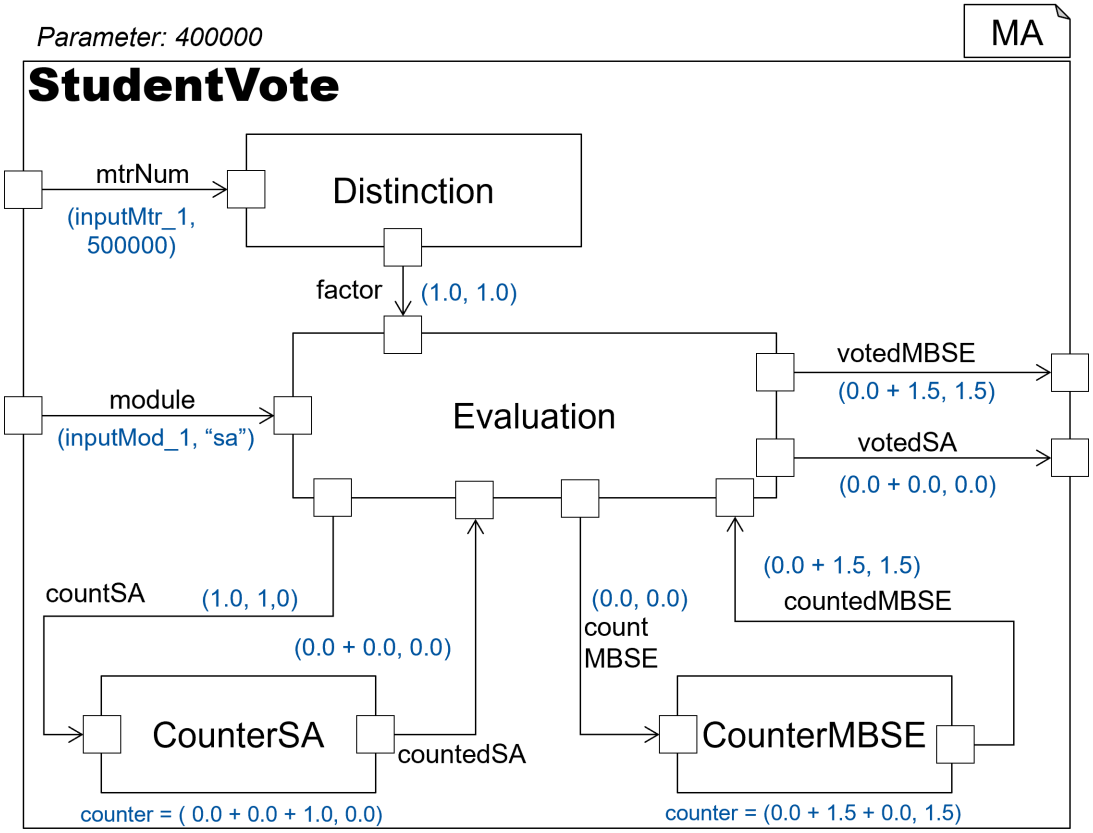

Figure 5: Symbolic and concrete representation of `StudentVote` internal state after the second input message

Information Gathered through Analysis During the execution of a model, various types of information are collected. This includes all possible transitions starting from the current state of the automaton, as well as all transitions that have been taken. For each transition, the collected information encompasses branch conditions and the unique name of the transition. The branch conditions specify the particular path taken through the model and must be satisfied by the input.

After processing each input message, information about the state of the automaton is gathered. This includes the current state's name, along with the symbolic and concrete values of internal variables. This data is collected for each sub-component and is subsequently combined into the state of the composite model.

All information regarding states and transitions is transmitted to the controller, where it can be utilized to implement termination conditions.

Functionality of a Controller A controller is responsible for implementing the execution strategy and the logic required for defining new inputs. To facilitate easy integration, each controller must adhere to a specific interface, allowing users the flexibility to implement the details according to their needs. The following paragraphs will elaborate on the general concept of a controller and its execution strategies, such as path coverage.

In fig. 6, an example automaton is presented. During model execution, each branch condition is collected and passed to the controller. Consider an input length of two; the solid blue arrows represents the first path taken. The controller is aware of the following information: `[A, -B, C, -D]`.

To achieve path coverage, an iterative negation of path constraints is employed. The first constraint is negated and provided to the solver, resulting in a new model that ensures a different path is taken. In the second recursion, the path represented by dashed orange arrows is taken, and `[-A, B]` is collected.

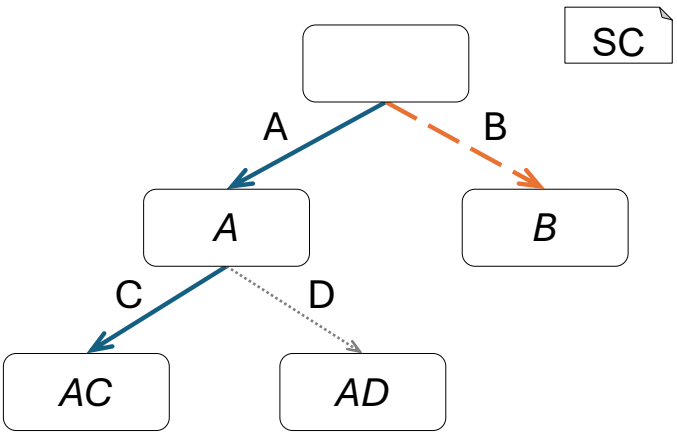

Figure 6: Example of an automaton to illustrate a controller where conditions A and D are satisfied by the input.

Next, B is negated. Since the combination `[-A, -B]` is unsatisfiable, the controller returns to the first recursion. There, the next constraint to be negated is C, as A and B were previously negated.

It is also possible to implement conditions within a controller to further control the paths taken; for example, ensuring that each transition is visited only once. In this case, the first and second recursions would remain the same as before. However, upon returning to the first recursion and checking `[A, -B, -C]`, the behavior would differ. Previously, a new path would be found and taken, but this path includes the transition with condition A, which has already been visited. As a result, the path execution is terminated, and the next possible path is explored.

In addition to branch conditions, the controller collects information regarding the state of the automaton. This information can be used to evaluate the controller or to validate the model, such as for path redundancy and non-determinism. These aspects are presented below.

We have developed multiple controllers, categorizing them into the following three groups:

Category 1: Path Coverage Controllers in the `Path Coverage` category aim to discover all possible paths within a model. Variations of these controllers implement oracles to handle non-deterministic models or utilize search algorithms.

Category 2: Termination Condition This category includes all controllers that utilize specified termination conditions. These conditions can be based on visited transitions or states, with or without involving internal variables. An example of such a termination condition can be found in section 5.2.

Category 3: Random Generation Controllers that do not utilize the symbolic information gathered by the tool fall under the category of *Random Generation*. This includes controllers that execute only once based on the given initial input or those that randomly generate input values that conform to the specified input types.

Path Redundancy Path redundancy arises when two distinct paths in a model exhibit identical simplified branching conditions. In the context of white box testing—where the internal workings and input-output behavior of individual components are analyzed—both paths are significant. Conversely, when the model is approached as a black box, both paths represent the same scenario; thus, only one occurrence is necessary to reflect the behavior of the system.

To identify path redundancy within a model, we examine the simplified representations of the branching conditions and outputs. If these simplified expressions are equivalent, we can conclude that a pair of redundant paths has been identified. For instance, consider the two paths: the first path has a

branch condition of `x + 1 < 5` and an output of `x + 1 - 2`. The second path consists of the branch condition `x + 2 < 6` and an output of `x + 2 - 3`. Upon simplifying these expressions for both paths, we find that they share the same branch condition of `x < 4` and an output of `x - 1`. This finding illustrates how path redundancy can simplify the analysis of models, ensuring efficiency in testing and validation processes.

Non-Determinism of the Model The extended generator supports models that include non-deterministic automata, which were previously unsupported by the code generator. In this context, non-determinism means that for a given input sequence, multiple paths and corresponding output sequences are permissible within the model.

Using the collected information, it is possible to determine whether a model is deterministic. Each path taken, represented by a set of branch conditions, is decomposed into symbolic expressions while preserving the order. For example, consider the following two paths:

The first path is represented by the condition $x > 5 \land y + 2 < 6$, where x and y are symbolic constants representing the input. The second path is represented by $x > 5 \land y + 2 < 8$.

Decomposing both paths yields the following representations:

$$[x > 5,\ y + 2 < 6] \text{ and } [x > 5,\ y + 2 < 8].$$

To assess the non-determinism of two paths, we compare the symbolic expressions at each corresponding position in their respective expression lists. Initially, we check each pair of expressions for equality. If they are found to be unequal, we then evaluate their satisfiability when combined through conjunction. If all pairs either contain equal expressions or yield satisfiable results when combined, we conclude that the two paths represent non-deterministic alternatives for some input values. Conversely, if any pair is unsatisfiable, it indicates that the paths are completely disjoint concerning the input.

In our specific example, the first pair `(x > 5, x > 5)` consists of equal expressions, while the second pair `(y + 2 < 6, y + 2 < 8)` comprises expressions that are satisfiable in conjunction. Consequently, these paths can be considered non-deterministic alternatives for certain values of x and y.

By employing this pairwise comparison approach across all paths, we can ascertain the number of non-deterministic paths. However, this method is computationally demanding and does not scale well with larger models, making it impractical for extensive applications. Furthermore, using an oracle to identify non-deterministic occurrences is not feasible, as such information is not mandatory. To address this limitation in larger models, we focus solely on detecting the presence of non-deterministic paths. Although this approach is less computationally intensive, it still necessitates multiple solver calls and comparisons, which can result in prolonged runtimes. Strategies for optimizing solver call usage to mitigate this issue are discussed in section 7.

5.3. Computation of Semantic Differences Between MontiArc Models

With the implementation of DSE in MontiArc, we can now compute semantic differences between two MontiArc models. To illustrate this process, we first introduce a modified version of the existing model `StudentVote`, referred to as `StudentVoteAlt`. Subsequently, we will explain how to compute semantic differences in the form of diff-witnesses.

The model `StudentVoteAlt`, as shown in fig. 7, retains the basic structure of `StudentVote` but introduces modifications to the `Evaluation` and `Counter` components. Notably, `Evaluation` 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.

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 fig. 7.

In this comparison, individual inputs are provided to each model for execution. For the first input, both models respond identically. However, the difference arises with the second input. When voting for `mbse&sa`, the `StudentVoteAlt` model determines the increment factor for its counter to be 2. At this point, the value of `CounterSA` remains unchanged, as it is less than 1.0. In contrast, `CounterMBSE` has a value of 1.0, triggering a reset to 0. This discrepancy will only manifest after the subsequent input, thus necessitating an input length of three to observe the difference. This example highlights a path that cannot be replicated in the original `StudentVote` model.

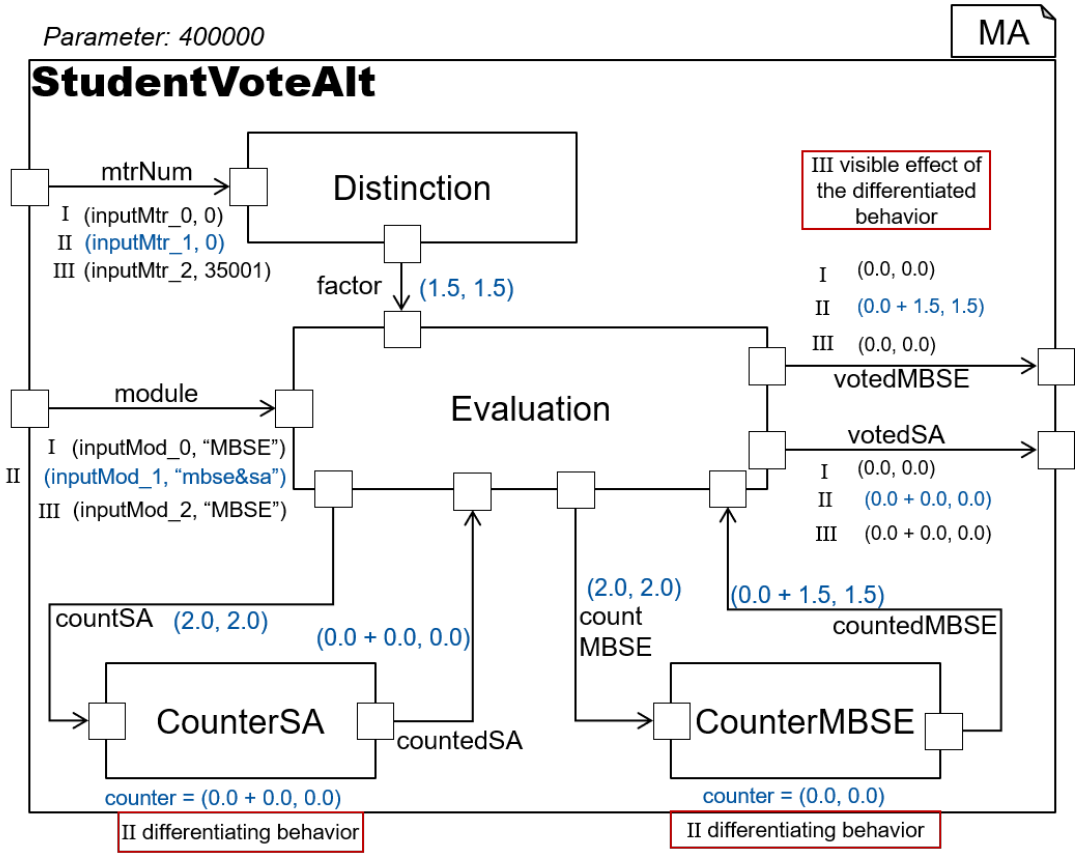

Figure 7: Example of semantic difference between `semDiffSmallModel` and `StudentVote`, input length 3

When assessing the semantic difference between two models, it is crucial to consider certain specific scenarios. One such scenario arises when a component in the second model contains an automaton that is deemed partial concerning the possible inputs of the first model. In this case, the partial automaton will respond to any unknown input by *ignoring* it—meaning it neither produces a new output nor transitions to a different state [39]. Consequently, the automaton retains its previous state and output. However, if the output ports are delayed, `Null` values may be generated. It is essential to address these cases within the automaton's definition of the component behavior.

The semantic difference is computed by comparing the outputs of both models while treating each model as a black box, thus eliminating the need to consider path conditions. The process is outlined in algorithm 1. The first model undergoes analysis through DSE, from which input-output pairs are extracted. These pairs are then utilized as inputs for the second model. The simplified symbolic portion of the computed output from the second model is compared to the corresponding output of the first model. If discrepancies are found between the simplified outputs, the input-output pair is classified as a diff-witness.

Until this point, we have implicitly assumed that the models under consideration are deterministic. However, our approach is also applicable to non-deterministic models. The non-determinism present in

the first component will be managed through DSE. In contrast, addressing the non-determinism in the second component requires an additional step following the initial output comparison. In this scenario, all potential paths must be explored to ensure the soundness of the differencing operator.

To achieve this, we utilize an oracle that represents or determines the non-deterministic "choices" within the model. All possible oracle values are computed and applied to the specific input. Each resulting output is then compared against the output of the first model. If a matching output is identified, the search is halted, and the comparison of output pairs continues. Conversely, if no matching output is found, the corresponding input-output pair from the first model qualifies as a valid diff-witness. Given that all potential paths through the model have been examined, the calculation of the semantic difference is deemed sound.

Algorithm 1 Algorithm to calculate semantic differences between MontiArc models

Input: m1, m2
Output: List of semantic differences

$diff\_cases \leftarrow [\,]$
$dse\_result \leftarrow execute\_DSE(m1)$
$found\_witness \leftarrow$ **False**

for each in_m1, out_m1 **in** dse_result **do**
  $out\_m2 \leftarrow runModel2(in\_m1)$
  $s\_out\_m1 \leftarrow simplify(out\_m1)$
  $s\_out\_m2 \leftarrow simplify(out\_m2)$

  if $s\_out\_m1 \neq s\_out\_m2$ **then**
    $found\_witness \leftarrow$ **True**
    $oracles \leftarrow calc\_oracles(m2, in\_m1)$

    for each oracle **in** oracles **do**
      $out\_oracle \leftarrow runModel2(in\_m1, oracle)$
      $s\_out\_oracle \leftarrow simplify(out\_oracle)$
      if $s\_out\_m1 == s\_out\_oracle$ **then**
        $found\_witness \leftarrow$ **False**
        break
      end if
    end for

  end if
  if $found\_witness$ **then**
    $diff\_cases \leftarrow diff\_cases \cup \{in\_m1, out\_m1\}$
  end if
end for
return $diff\_cases$

6. Evaluation

In this section, we evaluate the tool and the implemented controllers using the `StudentVote` example introduced in section 4 as our case study. We will first define the evaluation criteria, followed by a presentation of the results, including the calculation of semantic differences between two MontiArc models.

6.1. Definition of Evaluation Criteria and Controllers

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

Runtime The runtime of the developed tool plays a crucial role in determining its usability. To facilitate comparison, the runtime of each controller is classified into one of the following categories:

- *Constant Runtime*: The runtime remains unchanged regardless of the input length.
- *Linear Runtime*: The runtime increases proportionally with the input length.
- *Exponential Runtime*: The runtime grows exponentially as the input length increases.

Among these categories, controllers exhibiting constant runtime are deemed most desirable, followed by those with linear runtime, while exponential runtime is considered the least favorable option.

Minimalism The set of interesting inputs collected during the analysis should be minimized to ensure that the outputs remain simplified. Due to Z3's non-deterministic behavior in selecting concrete values for a given formula, we focus exclusively on symbolic values. However, simplification of each symbolic output results in the loss of information regarding the paths traversed, effectively treating the model as a black box.

For instance, consider two interesting inputs characterized by the following properties: the first input has a symbolic output of `x + 1 - 4` and a branch condition of `x + 1 < 5`, while the second input has a symbolic output of `x - 5 + 2` and a branch condition of `x - 5 < 10`. Here, x represents the symbolic representation of the input. Both inputs were derived from different paths, but when both symbolic outputs are simplified, they yield `x - 3`. Consequently, these two inputs would be considered duplicates concerning the *minimality* criterion.

As a metric, *minimalism* assesses the number of duplicates in relation to the size of the set of interesting inputs. Controllers are evaluated based on the percentage of duplicates, with a higher proportion of duplicates being regarded as less favorable.

Completeness When employing automata as the behavioral description of components, the completeness of a controller can be assessed from multiple perspectives. One perspective focuses on the number of visited transitions, represented as the ratio of visited transitions to the total number of existing transitions. A similar approach is used for assessing the completeness of states, which is determined by the ratio of visited states to the total number of existing states.

There are two perspectives regarding states: the first considers states without any information about the automaton's internal variables, while the second incorporates the current values of these internal variables as part of the state representation.

Given that the state space is often infinite concerning internal variables, it is not always feasible to calculate the percentage of visited states. Therefore, in such cases, the absolute number of visited states and transitions is used as an alternative metric for completeness.

6.2. Results of the Evaluation

In this subsection, we present the evaluation of the controllers using the model `StudentVote` from our motivating example as a case study. The implications of these results are discussed in section 7. It is important to note that all calculations and runtime measurements are specific to the hardware utilized during the evaluation. The evaluation was conducted on a ThinkPad T14s equipped with 32GB of RAM, an AMD Ryzen 7 Pro processor running at 2.7 GHz, and featuring 8 cores.

The detailed results of the evaluation are presented in fig. 8. In this visualization, poor performance is indicated in red, while good performance is marked in green. The results for the categories `Runtime` and `Completeness` are classified as follows: a percentage below 25% is deemed poor, while a percentage above 75% is considered good, with values in between categorized as neutral. Conversely, in the category

	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 8: Results regarding the smallModel based on input length 4

`Minimalism`, results above 75% are regarded as deficient, whereas those below 25% are classified as proficient.

Runtime The controllers categorized under `Path Coverage` experience exponential runtimes. For instance, an input length of 6 is estimated to take approximately 1.9 days to process. This exponential behavior can be attributed to the number of solver calls, which are computationally intensive and contribute significantly to the overall runtime, even for relatively small models and short input lengths.

In contrast, for the category `Termination Condition`, no uniform runtime can be established. When the termination condition is defined based on transitions, the runtime remains constant. However, when it relies on states—whether internal states are included or not—the runtime becomes linear. An exception arises with the `Boring Interesting Controller`, which utilizes specific states and transitions for its termination condition, leading to an exponential runtime.

The variability in runtime within this category can be explained by the nature of the termination conditions. For instance, if a condition is based on the frequency of visits to a particular transition in the automaton, the number of recursive calls is limited by that frequency, effectively creating a bottleneck. Similarly, state-based termination conditions can also act as bottlenecks. Moreover, controllers of the `Boring Interesting` type may suffer from poorly categorized transitions or states. Ineffectively chosen termination conditions can lead to situations where a transition is erroneously classified as interesting, allowing for numerous visits—even if that transition is never actually executed—resulting in the termination condition failing to apply.

Controllers in the category `Random Generation` exhibit a constant runtime. This consistency can be attributed to the non-utilization of collected symbolic information and the implementation of a cap on the maximum number of identifiable paths, which prevents infinite runtimes. The controller `RunOnce` executes the component precisely once for the specified input length, while `RandomInput` explores new paths exactly ten times using randomized inputs.

When assessing controllers based on runtime performance, those in the `Random Generation` category emerge as the most efficient, followed by those in the `Termination Condition` category. In contrast, the `Boring Interesting` controllers and those in the `Path Coverage` category are the least efficient, as they experience exponential runtimes.

Minimalism The architecture of the `StudentVote` system allows for multiple inputs to yield the same output. Initially, irrespective of the input provided, the output consistently appears as zero for both counters. This behavior is attributed to the previously discussed delayed ports. For input lengths of 3, the percentage of duplicate outputs ranges from 86% to 92%, while for input lengths of 4, this range increases to 91% to 99%. Notably, exceptions arise with the controllers `Termination Transition`,

Termination Automaton State, Termination State, and RunOnce. The RunOnce controller uniquely identifies a single path within the model, resulting in the absence of duplicates, regardless of input length. In contrast, the other exceptions can be attributed to a reduced number of interesting inputs discovered due to certain restrictions. Importantly, none of the evaluated controllers employ a strategy to prevent the generation of duplicate interesting inputs in relation to the simplified outputs.

Completeness of Visited Transitions To achieve completeness, it is essential that each transition is visited at least once. Controllers belonging to the categories Path Coverage and Termination Condition attain a transition coverage of 54% (7 visited transitions) for an input length of 1. However, a minimum input length of 3 is required to achieve full transition coverage of 100% (13 visited transitions). The Termination Transition controller is an exception; it is constrained by a limit on the number of visits allowed for each transition, resulting in a bottleneck for the initial two transitions in the StudentVote scenario.

In the case of controllers in the Random Generation category, transition coverage ranges from 31% to 46% (RunOnce: 4 or 5 visited transitions; RandomInput: 5 or 6 visited transitions), regardless of input length. This limited coverage is attributed to the restricted number of iterations performed by the controllers, which consequently reduces the number of paths explored.

Overall, all controllers are constrained by the specified input length when it comes to transition coverage. An inadequate input length hinders the complete detection of transitions. Conversely, when a sufficient input length is provided, controllers in the Path Coverage category can achieve full transition coverage. While controllers in the Termination Condition category may have the potential to explore all paths, they are ultimately restricted by their respective termination conditions.

Completeness of Visited States Excluding Internal Variables Controllers classified under Path Coverage and Termination Condition successfully visit 100% (6 states) of the reachable states starting from an input length of 1. In contrast, controllers categorized as Random Generation only manage to visit 67% (4 states) of the states, irrespective of the input length.

Completeness of Visited States An automaton's state can encompass the potential concrete values of all its internal variables. Consequently, StudentVote does not possess a finite state space due to its counter components. To quantify the percentage of visited states within the model, we relate it to the maximum number of states that can be reached based on the input length.

Controllers in the Path Coverage category consistently achieve 100% relative coverage, regardless of input length, except for the RandomNegation controller, which attains only 33% for an input length of 4. In the Termination Condition category, the percentage of visited states declines as input length increases, resulting in a completion rate of 10% to 40% for an input length of 4, largely due to the constraints imposed by their respective termination conditions. Controllers in the Random Generation category achieve merely 6% coverage for an input length of 4.

The ability to achieve completeness in visited states is contingent on both the model and the specified input length. Shorter input lengths are inadequate for attaining completeness in visited states for larger models.

Upon comparing the results, it becomes evident that no single controller emerges as the superior choice. The selection of a controller should be guided by the specific model in use and the objectives of the user. Notably, for controllers categorized under Path Coverage, an exponential increase in runtime is to be anticipated, regardless of the underlying model. If runtime efficiency is a primary concern and only a specific subset of paths is needed, opting for a controller of the Termination Condition type may prove to be the most effective strategy.

Most controllers, particularly those focused on path coverage, experience exponential growth in runtime. This phenomenon is exacerbated by the path explosion, which leads to a corresponding exponential increase in the number of solver calls and solver operations. The average CPU load on a single core due to these SMT-Solver operations is approximately 73 percent, indicating a pressing

need for optimization in solver operations. One effective optimization strategy involves implementing a timeout for the solver. If the solver fails to identify a solution within the allotted time, we can infer that the branch condition for the potential path is likely unsatisfiable, allowing it to be skipped.

A critical challenge in this optimization process lies in selecting an appropriate timeout duration. The goal is to maximize time improvement while minimizing result degradation. Here, result degradation is defined as the ratio of interesting inputs identified with the timeout versus those found without it.

To assess the time improvement and result degradation associated with the `Path Coverage Controller` in conjunction with garbage collection, various timeout durations were tested. The findings are summarized in table 1. An optimal balance between time improvement and result degradation appears to be achieved with a timeout set at 10 milliseconds. Shorter timeouts can lead to a degradation of results by as much as 99%, while longer timeouts yield only a modest time improvement of up to 14%.

Timeout in ms	Time improvement	Result deterioration
1	0.9970	0.9994
5	0.9965	0.9988
7	0.98	0.97
8	0.88	0.83
9	0.70	0.59
10	0.15	0.03
30	0.14	0.00
300	0.18	0.00
1000	0.17	0.00

Table 1
Different timeouts for input length three, controller: PC Garbage Collector

In conclusion, implementing a timeout for the solver can lead to significant runtime improvements. However, it is crucial to consider the accompanying loss of results. To achieve the optimal balance between time improvement and result degradation, the timeout must be individually calibrated for each combination of model, controller, and input length.

Computation of Semantic Differences Between MontiArc Models The evaluation of various controllers reveals that no single controller stands out as universally superior. Instead, the selection of a controller should be tailored to the specific needs of the user. Any controller can be employed to calculate the semantic difference; however, if the goal is to compute all diff-witnesses, it is recommended to select a controller from the `Path Coverage` category. Conversely, if identifying just a single diff-witness is sufficient, a category 2 controller may be more appropriate.

In the subsequent analysis, we compute the semantic difference using the `Path Coverage Controller` with a garbage collection trigger. The results of the semantic difference analysis between `StudentVote` and `smallModelSemDiff` for varying input lengths are presented in table 2.

Input length	Time in 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 2
Results of the computation of semantic differences between smallModel and semDiffSmallModel

As anticipated, a diff-witness was only identified starting from an input length of three. The challenges associated with calculating semantic differences mirror those of the selected controller. To analyze larger modules or to accommodate greater input lengths, optimization of the tool is necessary. Various optimization strategies are explored in section 7.

7. Discussion

As discussed in section 6, the runtime of the tool is significantly constrained by the CPU load associated with SMT solver operations. Specifically, the number of server calls increases exponentially with both the length of the input and the size of the model.

To address this issue, we have implemented a timeout strategy for the SMT solver as an optimization approach. The underlying assumption is that if no model is found within a specified timeframe, it is likely that no model exists. This strategy provides a dual benefit: it improves runtime while potentially compromising the quality of the results. Striking an optimal balance between these two factors necessitates selecting an appropriate timeout based on the specific controller in use. An alternative solution could involve parallelizing the analysis; however, the documentation on parallelism is limited, and its implementation for Z3 is not yet complete.

While the strategies presented offer some avenues for improvement, none provide a definitive solution to the runtime limitations. To substantially mitigate this issue, it is essential to reduce the number of solver calls. Achieving this requires the formulation of an effective controller strategy tailored to the specific use case. The challenge lies in minimizing solver calls while maximizing the discovery of paths.

In section 6, we evaluated three categories of controllers, each encompassing various implementations. While each category presents its own set of advantages and disadvantages, none emerged as the unequivocal best option. It is important to note that achieving comprehensive path coverage may necessitate accepting exponential runtime. Conversely, by settling for less exhaustive results, it is possible to attain a linear or constant runtime in relation to the input length.

To achieve an optimal balance between completeness and runtime, we propose a combination of controllers. Random input generation yields the least complete results but maintains a constant runtime. In contrast, a path coverage controller offers the most comprehensive results, albeit at the cost of exponential runtime. By integrating both controllers, we can likely attain an acceptable compromise regarding both completeness and efficiency.

One potential combination could be inspired by the queen's problem. Initially, a random input is provided to the model. Utilizing the symbolic information gathered from this input, the system then searches for a defined number of new paths. By generating a new random input and repeating this process, we can facilitate a more thorough exploration of the path tree within a user-defined search range. An example of the paths identified within such a tree is illustrated in fig. 9. Future evaluations will be necessary to determine whether this combined controller outperforms the existing implementations. In order to enable the integration of different controllers, modifications to the current controller architecture are required. This approach holds promise for significantly reducing the runtime of DSE, though the completeness of the results produced by the new controllers will also need to be assessed.

Another avenue for enhancing the tool is through decomposition analysis of models, as introduced by Godefroid [25]. This technique involves breaking down the program under analysis into individual functions. The results from analyzing these functions are then synthesized, taking into account their pre- and post-conditions, to evaluate composite programs. This methodology should be applicable to MontiArc models by examining their atomic components, which are already represented by atomic model artifacts. Implementing this approach will necessitate adjustments to the tool architecture.

Additional optimization strategies to consider include *fast unsatisfiability checks*, *common sub-constraint elimination*, and *incremental solving*, as utilized by jCUTE and discussed in section 3.

Optimization is a critical factor in analyzing realistic models. Our tool is currently in its initial version, and further optimization is essential for practical applications. At present, it supports primitive data types, strings, and enums for MontiArc models. Given that this represents only a fraction of the potential data types, expanding support for additional data types should be prioritized in future work.

A noteworthy observation regarding other DSE tools is that most utilize interpreters for Dynamic Symbolic Execution. However, there is currently no interpreter available for MontiArc models. Developing an interpreter for MontiArc poses several challenges, such as ensuring synchronization and managing the non-sequential execution of all components. For the purposes of this work, we deter-

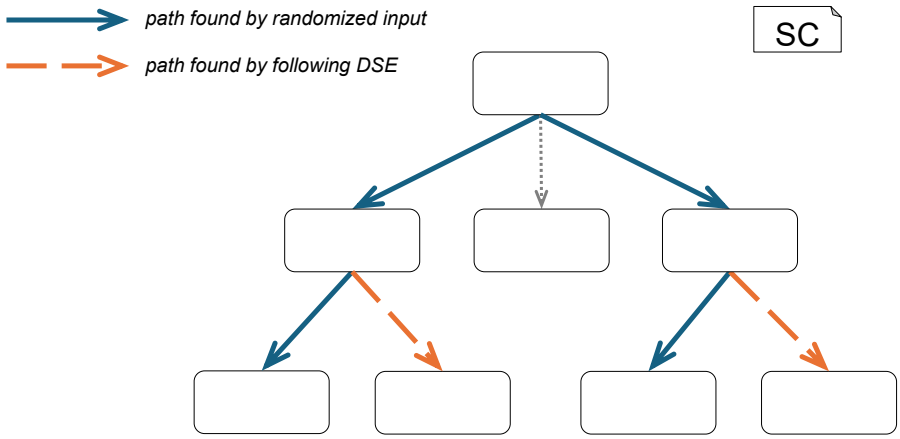

Figure 9: Possible paths found by a controller combining randomized inputs and DSE

mined that enhancing the MontiArc-to-Java generator would be sufficient. Nevertheless, introducing an interpreter would improve usability, as it would eliminate the need for generating additional files.

8. Conclusion

In this paper, we successfully developed a Dynamic Symbolic Execution (DSE) approach tailored for the component-and-connector architecture language MontiArc, implementing multiple execution strategies. This DSE framework enabled us to create a semantic differencing operator capable of detecting behavioral differences between two component-and-connector architectures. Notably, our approach addresses the limitations of previously developed semantic differencing operators by accommodating an infinite state space and input-output alphabet.

We evaluated our DSE approach and the implemented controller based on the criteria of *runtime*, *minimality*, and *completeness*, identifying scalability as the most significant challenge. While our semantic differencing approach is sound, it remains constrained by input length and does not scale well.

Looking ahead, we aim to tackle these challenges using the methods discussed in section 7, such as parallelization and strategic early evaluation cessation. We will also reevaluate the updated tool using existing component-and-connector models from both industry and scientific literature.

Furthermore, semantic differencing is not the only application of DSE concerning MontiArc models that interests us. Future work will explore DSE's applicability for test-case and input generation in MontiArc. We are also keen to investigate the highlighting of syntactic differences that lead to semantic differences, expand support for additional data types, and assess the use of an interpreter in comparison to our current generator-based approach. Finally, we may consider applying a DSE-based approach to other types of executable models.

Acknowledgments

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

References

[1] S. Maoz, J. O. Ringert, B. Rumpe, A Manifesto for Semantic Model Differencing, in: Proceedings Int. Workshop on Models and Evolution (ME'10), LNCS 6627, Springer, 2010, pp. 194–203.

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

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

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

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

[6] J. O. Ringert, B. Rumpe, M. Stachon, On Implementing Open World Semantic Differencing for Class Diagrams, Journal of Object Technology (JOT) 22 (2023) 2:1–14. doi:10.5381/jot.2023.22.2.a11.

[7] I. Drave, O. Kautz, J. Michael, B. Rumpe, Semantic Evolution Analysis of Feature Models, in: T. Berger, P. Collet, L. Duchien, T. Fogdal, P. Heymans, T. Kehrer, J. Martinez, R. Mazo, L. Montalvillo, C. Salinesi, X. Tërnava, T. Thüm, T. Ziadi (Eds.), International Systems and Software Product Line Conference (SPLC'19), ACM, 2019, pp. 245–255.

[8] A. Butting, O. Kautz, B. Rumpe, A. Wortmann, Semantic Differencing for Message-Driven Component & Connector Architectures, in: International Conference on Software Architecture (ICSA'17), IEEE, 2017, pp. 145–154.

[9] A. Butting, O. Kautz, B. Rumpe, A. Wortmann, Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution, Journal of Systems and Software (JSS) 149 (2019) 437–461.

[10] I. Drave, R. Eikermann, O. Kautz, B. Rumpe, Semantic Differencing of Statecharts for Object-oriented Systems, in: S. Hammoudi, L. F. Pires, B. Selić (Eds.), Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19), SciTePress, 2019, pp. 274–282.

[11] O. Kautz, Model Analyses Based on Semantic Differencing and Automatic Model Repair, Aachener Informatik-Berichte, Software Engineering, Band 46, Shaker Verlag, 2021.

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

[13] A. Haber, MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems, Aachener Informatik-Berichte, Software Engineering, Band 24, Shaker Verlag, 2016.

[14] A. Butting, O. Kautz, B. Rumpe, A. Wortmann, Architectural Programming with MontiArcAutomaton, in: In 12th International Conference on Software Engineering Advances (ICSEA 2017), IARIA XPS Press, 2017, pp. 213–218.

[15] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, W. Visser, Symbolic execution for software testing in practice: preliminary assessment, in: 2011 33rd International Conference on Software Engineering (ICSE), 2011, pp. 1066–1071. doi:10.1145/1985793.1985995.

[16] J. C. King, Symbolic execution and program testing, Commun. ACM 19 (1976) 385–394. doi:10.1145/360248.360252.

[17] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel, MontiCore 1.0: Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen, Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, 2006.

[18] K. Hölldobler, O. Kautz, B. Rumpe, MontiCore Language Workbench and Library Handbook: Edition 2021, Aachener Informatik-Berichte, Software Engineering, Band 48, Shaker Verlag, 2021.

[19] H. Giese, B. Rumpe, B. Schätz, J. Sztipanovits (Eds.), Science and Engineering of Cyber-Physical Systems (Dagstuhl Seminar 11441), Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[20] M. Broy, B. Rumpe, Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung, Informatik-Spektrum 30 (2007) 3–18.

[21] D. Harel, Statecharts: a visual formalism for complex systems, Science of Computer Programming 8 (1987) 231–274. doi:https://doi.org/10.1016/0167-6423(87)90035-9.
[22] B. Rumpe, Modeling with UML: Language, Concepts, Methods, Springer International, 2016.
[23] M. Broy, K. Stølen, Specification and development of interactive systems: focus on streams, interfaces, and refinement, Springer Science & Business Media, 2012.
[24] L. De Moura, N. Bjørner, Z3: An efficient smt solver, in: International conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2008, pp. 337–340.
[25] P. Godefroid, Compositional dynamic test generation, in: ACM-SIGACT Symposium on Principles of Programming Languages, 2007.
[26] X. Xiao, S. Li, T. Xie, N. Tillmann, Characteristic studies of loop problems for structural test generation via symbolic execution, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013, pp. 246–256. doi:10.1109/ASE.2013.6693084.
[27] K. Sen, D. Marinov, G. Agha, Cute: A concolic unit testing engine for C, volume 30, 2005, pp. 263–272. doi:10.1145/1095430.1081750.
[28] P. Godefroid, N. Klarlund, K. Sen, Dart: Directed automated random testing, in: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005, pp. 213–223.
[29] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, I. Finocchi, A survey of symbolic execution techniques, ACM Computing Surveys (CSUR) 51 (2018) 1–39.
[30] P. Godefroid, M. Y. Levin, D. A. Molnar, Automated whitebox fuzz testing, in: Network and Distributed System Security Symposium, 2008.
[31] N. Williams, Towards exhaustive branch coverage with pathcrawler, CoRR abs/2105.05517 (2021). arXiv:2105.05517.
[32] K. Sen, G. Agha, Cute and jcute: Concolic unit testing and explicit path model-checking tools, in: T. Ball, R. B. Jones (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 419–423.
[33] M. Mues, F. Howar, Jdart: Dynamic symbolic execution for java bytecode (competition contribution), in: A. Biere, D. Parker (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer International Publishing, Cham, 2020, pp. 398–402.
[34] K. Havelund, T. Pressburger, Model checking java programs using java pathfinder, International Journal on Software Tools for Technology Transfer 2 (2000) 366–381.
[35] W. Visser, C. S. Păsăreanu, S. Khurshid, Test input generation with java pathfinder, in: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, 2004, pp. 97–107.
[36] K. Luckow, M. Dimjasevic, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamaric, V. Raman, Jdart: A dynamic symbolic analysis framework, in: M. Chechik, J.-F. Raskin (Eds.), Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 9636 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 442–459. doi:10.1007/978-3-662-49674-9_26.
[37] S. Maoz, J. O. Ringert, B. Rumpe, An Operational Semantics for Activity Diagrams using SMV, Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, 2011.
[38] N. Bjørner, L. de Moura, L. Nachmanson, C. M. Wintersteiger, Programming z3, Engineering Trustworthy Software Systems: 4th International School, SETSS 2018, Chongqing, China, April 7–12, 2018, Tutorial Lectures 4 (2019) 148–201.
[39] B. Rumpe, Formale Methodik des Entwurfs verteilter objektorientierter Systeme, Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.