SIMPOSE: Configurable N-Way Program Merging Strategies for Superimposition-based Analysis of Variant-Rich Software

Dennis Reuling, Udo Kelter
Software Engineering Group
University of Siegen, Germany
{dreuling | kelter}@informatik.uni-siegen.de

Sebastian Ruland, Malte Lochau
Real-Time Systems Lab
TU Darmstadt, Germany
{sebastian.ruland | malte.lochau}@es.tu-darmstadt.de

Abstract—**Modern software often exists in many different, yet similar versions and/or variants, usually derived from a common code base (e.g., via clone-and-own). In the context of product-line engineering, family-based analysis has shown very promising potential for improving efficiency in applying quality-assurance techniques to variant-rich software, as compared to a variant-by-variant approach. Unfortunately, these strategies rely on a product-line representation superimposing all program variants in a syntactically well-formed, semantically sound and variant-preserving manner, which is manually hard to obtain in practice. We demonstrate the SIMPOSE methodology for automatically generating superimpositions of N given program versions and/or variants facilitating family-based analysis of variant-rich software. SIMPOSE is based on a novel N-way model-merging technique operating at the level of control-flow automata (CFA) representations of C programs. CFAs constitute a unified program abstraction utilized by many recent software-analysis tools. We illustrate different merging strategies supported by SIMPOSE, namely variant-by-variant, N-way merging, incremental 2-way merging, and partition-based N/2-way merging, and demonstrate how SIMPOSE can be used to systematically compare their impact on efficiency and effectiveness of family-based unit-test generation. The SIMPOSE tool, the demonstration of its usage as well as related artifacts and documentation can be found at *http://pi.informatik.uni-siegen.de/projects/variance/simpose*.**

Index Terms—**Program Merging, Model Merging, Software Testing, Family-Based Analyses**

I. INTRODUCTION

In this contribution, we demonstrate a novel methodology and accompanying tool support, called SIMPOSE, for automatically constructing superimpositions of N given variants and/or versions of C programs. The resulting representation facilitates seamless subsequent applications of recent family-based analysis techniques and tools from product-line engineering. Those techniques allow for efficiently analyzing whole families of programs in a single analysis run instead of considering every variant one-by-one [1]. To this end, SIMPOSE employs a novel N-way model-merging technique [2] being applicable to the control-flow automata (CFA) representation of the given input programs. The resulting superimposed CFA is syntactically well-formed, semantically sound and variant-preserving by utilizing the concept of variability encoding. We further show how SIMPOSE supports configurable merging strategies, namely variant-by-variant, N-way merging, incremental 2-way merging, and partition-based N/2-way merging and we demonstrate how SIMPOSE can be used to systematically compare their impact on efficiency and effectiveness of family-based unit-test generation [3]. The methodology presented in this contribution is based on a previous work, recently published in Reuling et al. [4].

II. BACKGROUND AND MOTIVATION

Control-Flow Automata. Program superimposition in SIMPOSE is based on an N-way model-merging technique applied to a graph-based representation of C programs, called *control-flow automata (CFA)* [5]. CFA nodes denote *program locations* and CFA edges denote *control-flow transfers* between locations. Edges are labeled with respective code fragments, either denoting basic blocks of value assignments to program variables, or conditional expressions over those variables. Hence, CFAs reflect essential control-flow structures together with the corresponding data flows thus capturing all possible *program execution paths*. As an example, the CFA in Fig. 1A corresponds to a C program unit with input variables x and y, output variable r and local variable a. The program assigns the greater value of x and y to a, doubles this value and returns the result.

Unit-Test Generation. Unit testing constitutes one of the most established quality-assurance techniques in practice for systemically investigating a program unit for potential bugs and faults. To this end, a set of *test cases* by means of exemplary input-variable value assignments is selected into a *test suite* to be executed on a program unit under test in order to compare the actual (and possibly faulty) output with the expected one. To control the number of test cases, code-coverage criteria (e.g., basic-block coverage or branch coverage) are often used, defining *test goals* in programs to be reached by the selected test cases. Applying basic-block coverage to the CFA in Fig. 1A imposes four test goals:

$$tg_1\text{: int a, } tg_2\text{: a} = \text{x, } tg_3\text{: a} = \text{y, and } tg_4\text{: r} = \text{a} * 2,$$

which are covered by at least two test cases, e.g.,:

- tc_1: $(x = 1, y = 0)$ covering tg_1, tg_2, tg_4, and

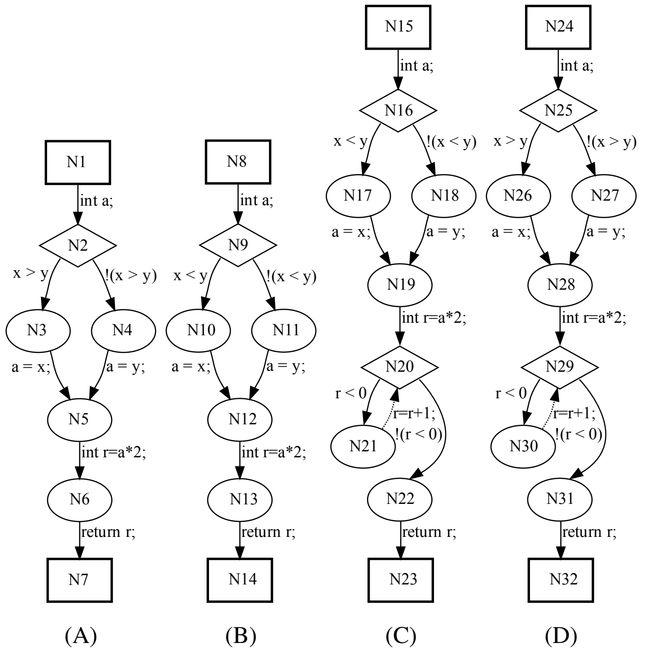

Fig. 1: Variants of C Programs (CFA Representation)

- tc_2: $(x = 0, y = 2)$ covering tg_1, tg_3, tg_4.

The aim of *automated test-suite generation* is to generate from a program under test a set of test cases satisfying a coverage criterion (i.e., reaching each test goal by at least one test case) [6]. If, in addition to the input program, also a functional specification of that program is available, test cases may be further enriched by expected output values (oracles) which would facilitate fully-automated test-execution and test-result evaluation of generated test suites. Automated test-generation methodologies may be evaluated by two (generally contradicting) criteria, namely *efficiency* and *effectiveness*.

Testing Efficiency. Efficiency may be measured in two ways:

1) computational effort for test-suite generation (e.g., CPU time for test-suite generation), and
2) computational effort for test-suite execution (e.g., number of generated test cases).

For instance, concerning criterion 2, we require at least two test cases for basic-block coverage of the program in Fig. 1A.

Testing Effectiveness. Effectiveness may be also measured in two ways:

1) number of test goals covered during test-suite generation, and
2) number of faults and bugs detected during test-suite execution.

For instance, concerning criterion 1, a test suite with tc_1 and tc_2 achieves 100% basic-block coverage on the program in Fig. 1A. In practice, coverage is often less than 100% due to time-outs during reachability analysis or non-reachability of test goals. Concerning criterion 2, either realistic or seeded faults (e.g., mutations) may be used for measurements.

Program Variants/Versions. So far, we were solely concerned with one specific *version* of one particular *variant* of a given program. Nevertheless, modern software consists of hundreds or even thousands of concurrent variants (e.g., due to a massive number of configuration options) and, additionally, undergo continuous evolution leading to consecutive program versions. In the following, we will not distinguish between program variants and versions, but consider both as different facets of *program variability*. As an example, assume the program in Fig. 1A to constitute initial variant A of a *family* of similar programs. Next, a developer may create variant B in Fig. 1B in which the smaller instead of the greater value of x and y is doubled. Hence, the developer copies A and simply flips the comparison operator in the if-condition to obtain B (so-called *clone-and-own* approach). Similarly, the two variants C and D in Figs. 1C and 1D, may be created, where C is derived from B and D is derived from C both by adding a code fragment for ensuring value r to be non-negative [1].

Test-Case Reuse. In order to also assure every member of such a program family by unit testing, each new program variant has to be covered by a sufficient set of test cases as described above. However, generating test suites for every new program variant in separate from scratch is often inefficient, yielding many redundant test-generation runs due to the presumably very high amount of similarity (shared program paths) among the different program variants. In case of larger program families, such a variant-by-variant approach may be even practically infeasible. Instead, it has been shown beneficial to reason about a possible *reuse* of test cases between variants [3]. For instance, tc_1 and tc_2, initially generated for covering variant A, can be *reused* for achieving basic-block coverage also on variant B (where, however, the expected outputs must be reversed). In contrast, for covering C and D, at least one additional test case (e.g., $tc_3 := (x = -1, y = -2)$) is required for reaching the new basic block $\mathsf{r} = \mathsf{r} + 1$.

Program Superimposition and Variability Encoding. In order to automatically reason about re-usability of generated test cases among variants, an integrated representation of similarity and variability between all members of a program family is required. However, such a *superimposed representation* [2] is usually not available a priori, but has to be (re-)constructed from the set of program variants. Fig. 2 (left) shows a superimposed CFA representation of all four variants A, B, C and D. Here, variant-specific CFA paths are marked by additionally introduced program branches guarded by *presence conditions* over fresh conditional variables associated with the respective variants. This construction is known as *variability encoding* [4]. For instance, the left branch marked by presence condition $A||D$ corresponds to the if-branch shared by the program variants A and D.

Family-based Analysis. It has been shown in recent research that a superimposed representation including presence conditions can be used for improving efficiency of quality assurance of program families [1]. A detailed discussion regarding

[1]Please note that this rather over-complicated code fragment is used only for illustrative purposes.

[2]Such representation comprises all program variants in a syntactically well-formed, semantically sound and variant-preserving way.

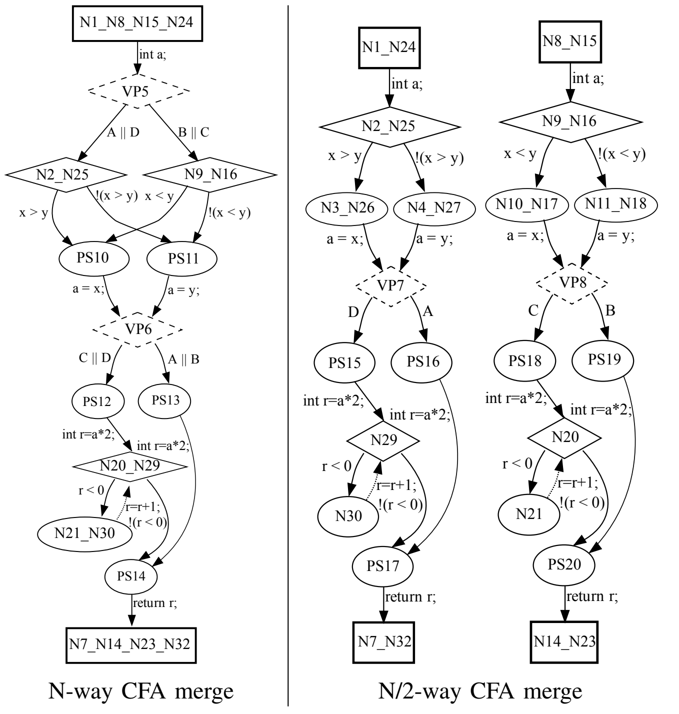

Fig. 2: Results of different Merging Strategies in SIMPOSE

application scenarios and constraints of family-based analyses can be found in [4]. For instance, the superimposed CFA in Fig. 2 (left) allows to directly generate a test-suite achieving full coverage of all program variants in a family-based manner instead of considering every variant one-by-one [3]. To this end, generated test cases also obtain a presence condition, being the conjunction of presence conditions visited along the respective CFA path traversed by the test case. Hence, this condition denotes the set of variants for which this test case is feasible. For instance, concerning basic-block coverage on all program variants superimposed in Fig. 2 (left) may yield the test suite:

- $tc_1 := (x = 1, y = 0)$ with presence condition $A||D$,
- $tc_2 := (x = 0, y = 2)$ with presence condition $B||C$, and
- $tc_3 := (x = -1, y = -2)$ with presence condition $C||D$.

For a superimposition-technique to be feasible for subsequent family-based analysis, it must ensure the resulting superimposition to be *syntactically well-formed*, *semantically sound* and *variability preserving* [4]. In addition, testing efficiency in terms of the size of test suites generated from a superimposed representation depends on the *precision* of the superimposed representation; the more precisely similar parts shared among different program variants are identified and integrated, the more test cases may be reused thus reducing the overall number of required test cases. For instance, if a less precise superimposition technique would not identify the similar if- and else-branches of A and D as well as of B and C, this may result in 4*3=12 test cases for basic-block coverage of all variants (instead of 3) thus leading a similar testing effort as a variant-by-variant approach.

III. THE SIMPOSE APPROACH

We now describe the SIMPOSE tool for automatically generating superimposed CFAs for a set of N C program variants, serving as input for family-based software-analysis tools. Our tool incorporates configuration options for different merging strategies and enables users to investigate their impact on analysis efficiency/effectiveness.

A. Overview

Figure 3 provides an overview of SIMPOSE. In the first step, N C program variants are transformed into CFA representations to serve as input models for our N-way CFA model-merging methodology. Our N-way model-merging algorithm consists of the three default stages *compare*, *match* and *merge* [2], followed by an additional stage for *variability encoding*. A detailed description of each stage can be found

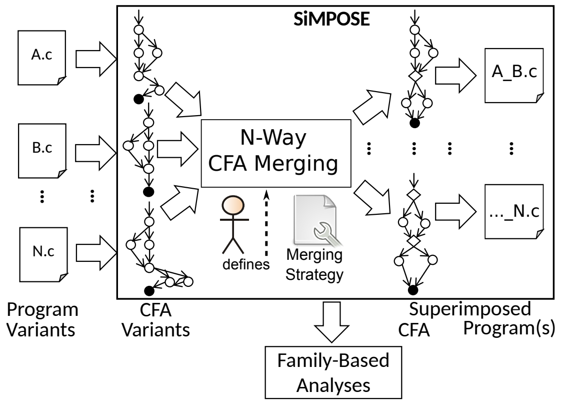

Fig. 3: SIMPOSE Overview and Usage Scenario.

in Reuling et al. [4]. As a result, SIMPOSE returns the superimposition either as CFA model or as C code which can be used as input for off-the-shelf family-based analysis tools (e.g., test-case generators or model-checkers) [1].

B. N-way Program Merging Strategies

The SIMPOSE tool provides several configuration options which allow users to experiment with freely adjustable *merging strategies* and to compare the results with respect to testing efficiency and effectiveness. Figure 4 shows statistics for exemplary results provided by SIMPOSE if applied to the BusyBox function `eject`[3] to support comparisons between different merging strategies. We first describe the variant-by-variant (i.e., non-merging) strategy to serve as a baseline.

Strategy 0 (Variant-by-Variant, VbV). SIMPOSE just returns the set of N input programs as N (non-superimposed) CFAs which are passed to a test-suite generator one-by-one.

Strategy 1 (2-Way Merging). The set of N input programs is merged *incrementally* into one superimposed CFA by performing $N-1$ consecutive 2-way CFA model-merging steps. To this end, SIMPOSE is able to take as input CFAs which already contain presence conditions from previously merging

[3]Taken from *https://busybox.net/* , version 1.24.2. All BusyBox functions and evaluation results are discussed in [4].

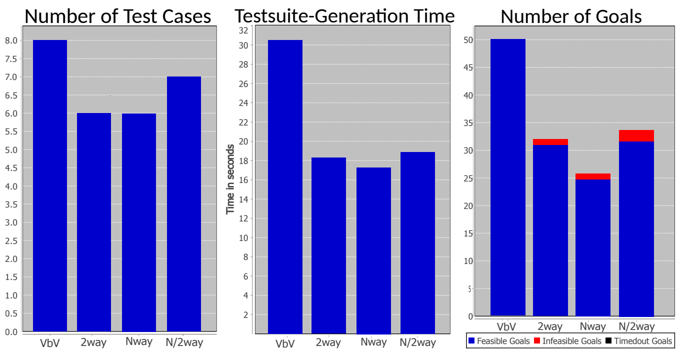

Fig. 4: Testing Efficiency and Effectiveness for BusyBox Function `eject` using different SIMPOSE Merging Strategies.

steps. After $N-1$ merging steps, the resulting CFA superimposes all N program variants and can be used as input for a single application of a family-based test-suite generator.

Strategy 2 (N-way Merging). The set of N input programs is merged *all at once* into one superimposed CFA by performing one N-way model-merging step in a single run. Similar to 2-way merging, the final superimposed CFA contains all N variants and can be used as input for a single application of a family-based test-suite generator.

Strategy 3 (N/2-way Merging). The set of N input programs is partitioned into an adjustable number of disjoint subsets (e.g., two subsets each containing $N/2$ variants). For each subset, a superimposed CFA is constructed in a single run (e.g., performing two $N/2$-way merges) and to each of the resulting superimposed CFAs, the family-based test-suite generator is applied separately. For instance, Fig. 2 (right) shows the result of applying $N/2$-way merging to our running example, where the left merge superimposes variants A and D and the right merge superimposes variants B and C.

Discussion. From the statistics provided by SIMPOSE, the user may gain the following insights.

- VbV is slightly more *effective* than the other strategies as it is the only approach to achieve 100% coverage on all variants. However, using N-way, we have to cover the smallest number of test goals as the superimposition is the most precise one, followed by 2-way and $N/2$-way.
- Concerning *efficiency* in terms of test suite size, N-way and 2-way both perform better than $N/2$-way, whereas VbV performs worst as expected.
- Concerning *efficiency* in terms of computational effort, all three merging strategies clearly perform better than VbV, where N-way shows the best overall performance.

Outlook. The SIMPOSE tool may be used for investigating the impact of further strategies (e.g., prioritization metrics for 2-way and different subset partitioning strategies for $N/2$-way). In addition, the presented strategies may be combined in arbitrary ways, especially in case of larger families of programs (e.g., partitions may be merged using 2-way and the result may be further merged using N-way or vice versa). Regarding applicability in other contexts, we plan a) to adapt SIMPOSE and accompanying family-based analyses tools to other programming languages and language concepts than C as well as b) to define precise merging operators for other complex modeling languages beyond CFA [7].

IV. Related Work

CFAs constitute a medium-grained abstraction used by many software-analysis tools [5]. Existing approaches for comparing CFA-like representations are usually used in reverse-engineering contexts [8]. Since SIMPOSE applies N-way model merging to CFA, our approach is also related to corresponding techniques (e.g., model versioning [9]–[11]). These approaches require conflicts to be explicitly resolved in an either-or-manner (i.e., by manual selections), whereas our methodology resolves all conflicts automatically. An abstract characterization of the N-way model merging problem is presented in [2] and applied to UML class diagrams and state machines. Martinez et al. also present a framework for extracting product-line models from a set of model variants [12]. In both works the authors introduce several operators which can be implemented in a model-specific way. All those existing approaches do not consider as an application scenario a subsequent family-based analysis as provided by SIMPOSE.

Acknowledgments

This work was partially supported by the DFG (German Research Foundation) within the CoMoVa project (grant nr 330452222). This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

References

[1] T. Thüm, S. Apel, C. Kästner, and G. Schaefer, I.and Saake, “A Classification and Survey of Analysis Strategies for Software Product Lines,” *ACM Comput. Surv.*, vol. 47, no. 1, pp. 6:1–6:45, Jun. 2014.

[2] J. Rubin and M. Chechik, “N-way Model Merging,” in *Proceedings of the 2013 Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 301–311.

[3] J. Bürdek, M. Lochau, S. Bauregger, A. Holzer, A. von Rhein, S. Apel, and D. Beyer, “Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines,” in *FASE*. Springer, 2015, pp. 84–99.

[4] D. Reuling, U. Kelter, J. Bürdek, and M. Lochau, “Automated N-way Program Merging for Facilitating Family-based Analyses of Variant-rich Software,” *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 3, pp. 13:1–13:59, Jul. 2019.

[5] D. Beyer and M. E. Keremoglu, “CPAchecker: A Tool for Configurable Software Verification,” in *Proc. CAV*. Springer, 2011, pp. 184–190.

[6] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar, “Generating tests from counterexamples,” in *Proceedings. 26th International Conference on Software Engineering*, 2004, pp. 326–335.

[7] D. Reuling, M. Lochau, and U. Kelter, “From Imprecise N-Way Model Matching to Precise N-Way Model Merging,” *Journal of Object Technology*, vol. 18, no. 2, 2019.

[8] P. P. F. Chan and C. Collberg, “A Method to Evaluate CFG Comparison Algorithms,” in *2014 14th International Conference on Quality Software*, Oct. 2014, pp. 95–104.

[9] K. Altmanninger, M. Seidl, and M. Wimmer, “A survey on model versioning approaches,” *International Journal of Web Information Systems*, vol. 5, no. 3, pp. 271–304, 2009.

[10] H. K. Dam, A. Egyed, M. Winikoff, A. Reder, and R. E. Lopez-Herrejon, “Consistent merging of model versions,” *Journal of Systems and Software*, vol. 112, pp. 137–155, 2016.

[11] C. Debreceni, I. Ráth, D. Varró, X. De Carlos, X. Mendialdua, and S. Trujillo, *Automated Model Merge by Design Space Exploration*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 104–121.

[12] J. Martinez, T. Ziadi, T. F. Bissyande, J. Klein, and Y. L. Traon, “Automating the Extraction of Model-Based Software Product Lines from Model Variants,” in *ASE 15*, Nov. 2015, pp. 396–406.