Electronic Communications of the EASST
Volume 84 Year 2025

12th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation / 2nd AISoLA - Doctoral Symposium, 2024

Edited by: Sven Jörges, Salim Saay, Steven Smyth

Towards Test-Driven Conflict Resolution

Jonas Schürmann, Bernhard Steffen

DOI: 10.14279/eceasst.v84.2677

Towards Test-Driven Conflict Resolution

Jonas Schürmann and Bernhard Steffen

Chair for Programming Systems, TU Dortmund University, Dortmund, Germany

{jonas.schuermann,steffen}@cs.tu-dortmund.de

Abstract: Merge conflicts often prove difficult to resolve because they break current versioning and development tools. Developers are forced to resolve all conflicts upfront, often alone, and without tool support, before they can start to explore the problems. Building on our lazy merging versioning concept, we introduce the paradigm of test-driven conflict resolution, which turns this process on its head: When merge conflicts arise, developers first explore the conflicted program with a lazy interpreter that interactively executes tests and debugs the conflicted program. Only after developing a sufficient understanding of the program's behavior do they begin making resolution decisions. In addition to interactive exploration, we use lazy interpretation for test-by-test conflict analysis and conflict resolution synthesis. Combining static analyses of the fine-grained version history, dynamic analyses during interpretation, and immediate feedback from test execution facilitates an effective resolution process.

Keywords: Collaborative Software Development, Distributed Version Control, Test-Driven Development, Test-First, Software Integration, Conflict Tolerance, Laziness, Conflict Resolution Synthesis

1 Introduction

Branching & merging workflows are ubiquitous because they provide developers in a team with isolated workspaces where they can work on their own tasks without interruption. Typically, developers use branches for cohesive tasks and merge their contributions into the main version of the software once they are finished [BBR+12]. Because branching & merging boosts the development team productivity by enabling concurrent development with minimal interference, it has become an essential part of modern development practices.

The downside to this increased freedom and productivity is the challenging task of integrating the concurrent contributions. Since developers are usually unaware of their colleagues' concurrent development efforts, they may introduce changes that contradicts their work. Finding good resolutions to these merge conflicts is a major challenge in the field of software versioning [Men02]. The accumulation of merge conflicts can even create an "integration hell" [PSW11] that is extremely difficult to resolve.

This problem is exacerbated by the fact that merge conflicts render most development tools inoperable: Interpreters, debuggers, test suites, and type checkers stop functioning. This leaves developers to work out resolutions "in the dark". The problem worsens as the number of merge conflicts increases because developers must specify resolutions for all conflicts before they can even begin to validate their choices or execute and debug the application. They are forced to make premature and uninformed resolution decisions in a cumbersome process of trial-and-error that provokes mistakes.

We support developers in this challenging situation by properly representing merge conflicts and enabling the execution and debugging of conflicted programs. By facilitating existing test suites, we establish the paradigm of test-driven conflict resolution.

In this paper, we make the following contributions:

- We present a lazy interpreter that can execute conflicted programs interactively. This interpreter operates on a new internal program representation that cohesively integrates abstract syntax, distributed history, and merge conflicts.
- We introduce a test-first approach to conflict resolution that uses dynamic exploration to foster understanding and improve resolution decisions.
- We demonstrate how test-driven conflict resolution can support automatic conflict resolution by speeding up the search for resolution candidates and by combining partial candidates to a good overall resolution.

Section 2 will first discuss the lazy merging of abstract syntaxes and the resulting internal program representation. In Section 3 we explain how this new internal representation enables the execution of conflicted programs. We will also discuss the new capabilities to support conflict resolution work. Section 4 demonstrates new test-based analysis approaches that our lazy interpreter enables. Next, Section 5 sketches the synthesis of conflict resolutions in more complex situations. Section 6 contrasts our approach with existing related work before the paper finishes with the conclusion in Section 7.

2 Lazy Merging

Lazy merging is our take on conflict-tolerant versioning. In contrast to current state-of-the-art versioning systems that enforce the immediate resolution of conflicts during merging, lazy merging cleanly separates merging from conflict resolution. It automatically integrates concurrent contributions and represents merge conflicts within the source syntax, to be resolved at a later time. This empowers developers to share conflicts to resolve them collaboratively, continue working in their presence, and record resolutions in explicit commits. Because lazy merging records and builds upon the causality relation of operations, we have been able to prove strong properties about its merging behavior using lattice theory. Specifically, we have demonstrated that unique merges can be easily computed with complete and minimal conflict representations, and that the merge operation is associative, commutative, and idempotent [SS23].

This paper will use the collaborative development of a simple point-and-click adventure as a guiding example. In the game, a player can explore a cottage in the forest, and two developers have different ideas for what is behind the cottage door. Figure 1 shows the original game model on the left, where the door leads to a normal room. Developer 1 (top) replaced the room with a treasure, while developer 2 (bottom) concurrently pointed the door onto a locked chest, planning to

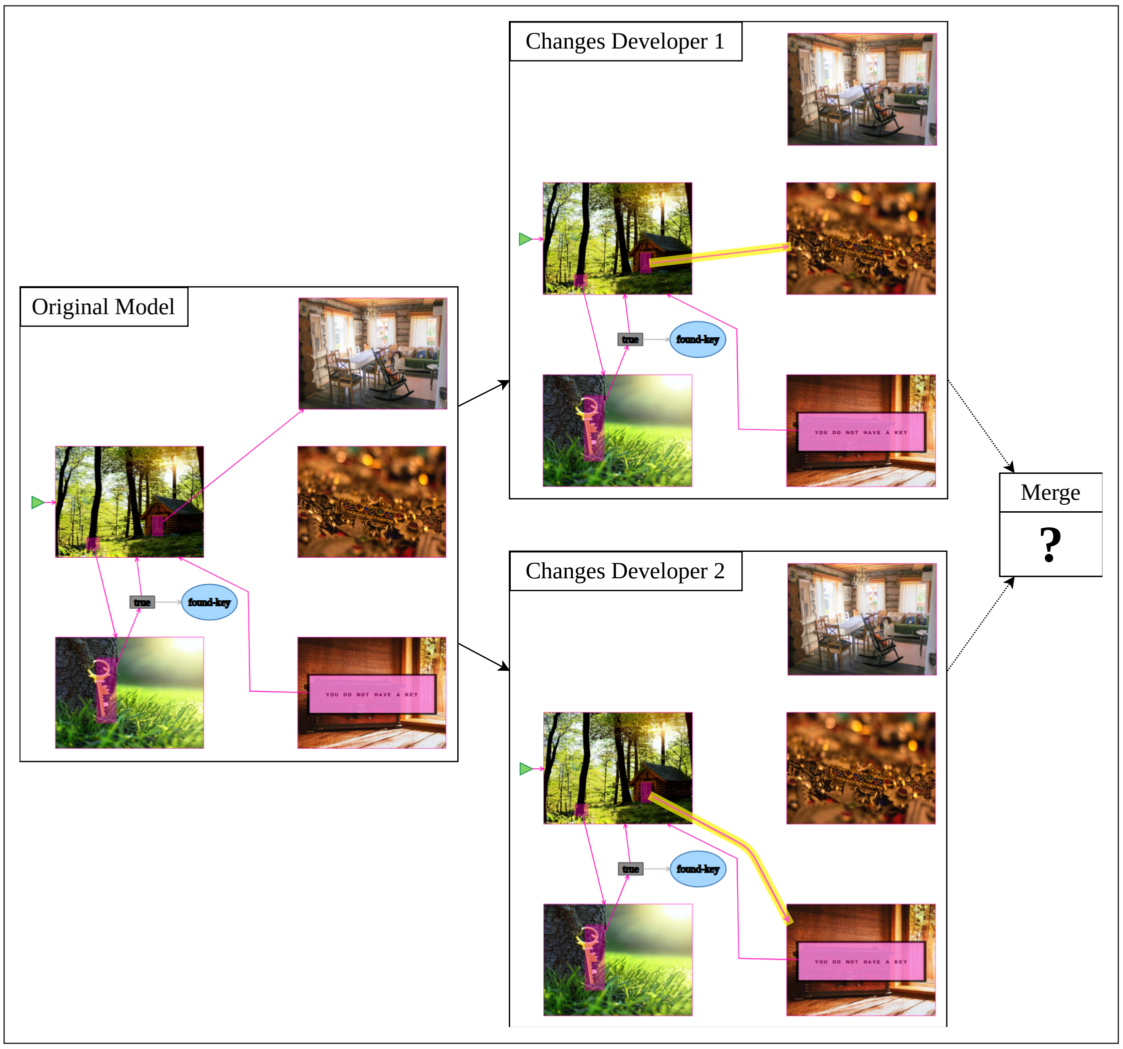


Figure 1: Collaboration scenario for the guiding example: Two developers provoke a conflict by concurrently moving the destination of the door click area to different screens.

later integrate the golden key into the story. The two developers want to merge their contributions, which faces them with the conflict in the cottage door destination. Now, they need to find a resolution that satisfies both of them.

Lazy merging applies a segmentation into uniquely identifiable cells onto the program representation. These cells are the atomic building blocks to which values can be assigned by edit operations. A cell's persistent identifier never changes, allowing concurrent assignments to be easily matched during merging. To realize this segmentation in a program's abstract syntax, each syntax element is assigned a persistent UUID, as well as cells for all of its properties. In the game model, each screen, click area, variable, and transition is identified by an internal UUID, through which all properties can be reliably accessed. Syntax elements record their relationships in cells as well, referencing the unique identifiers of other elements. Moving the starting point of an edge to a different node records a new assignment in the `sourceID` cell of the edge entity. After merging, the cells collect all their concurrent assignments and identify possible conflicts.

Figure 2 illustrates how lazy merging versions the abstract syntax of a point-and-click adventure. First and foremost, the causal order of all assignments is recorded. Then, cartesian lifting decomposes this order across all cells, giving each cell the causal order of its own assignment operations. The leaves of the cell's own causal order determine the valuation and show the impact of global concurrency on the state of each syntactic element. Concurrent assignments to the same cell create multiple leaves in the cell's causal order, indicating a conflict between these values. The persistent segmentation of cells maintains the overall syntactic structure intact and precisely locates edit conflicts.

The abstract syntax is easily constructed from the cell decomposition valuations. In the example, circles represent screens that the player can navigate by selecting transitions between them. The labels of screens and transitions, as well as the source and target relationships of transitions, are stored in cells. When cells are multivalued, a conflict operator representing all possible choices is inserted into the abstract syntax. The grammar of the abstract syntax is extended with conflict operators for all properties and relationships of syntax elements.

Lazy merging operates on the abstract syntax to prevent presentational details from interfering in the merge process. For this reason, lazy merging requires a projectional editor to render the internal representation into views of human-readable surface syntax. The views present the program in graphical or textual notations and propagate edits back into the abstract syntax. As the projectional views keep track of the unique identifiers in the syntax, changes are precisely recorded and matched. Figure 3 shows graphical and textual surface syntaxes for the point-and-click adventure specification that visualize conflict operators differently. The graphical syntax introduces a new lightning node that indicates a conflict in the edge connection and visualizes all options with additional edge segments. The textual syntax uses a block element to list all options of the ambiguous reference.

We currently use our Cinco Cloud language workbench [BBK+22] to realize these projectional editors for domain-specific graphical languages. Cinco Cloud provides projectional structure editors that are easily attached to this new abstract syntax. Cinco Cloud's modular implementation makes visualizing and interacting with conflicts straightforward and allows us to apply our concept to many graphical languages at once.

However, our approach is not limited to graphical programming languages. It can also be applied to mainstream software development where general-purpose languages and textual notations based

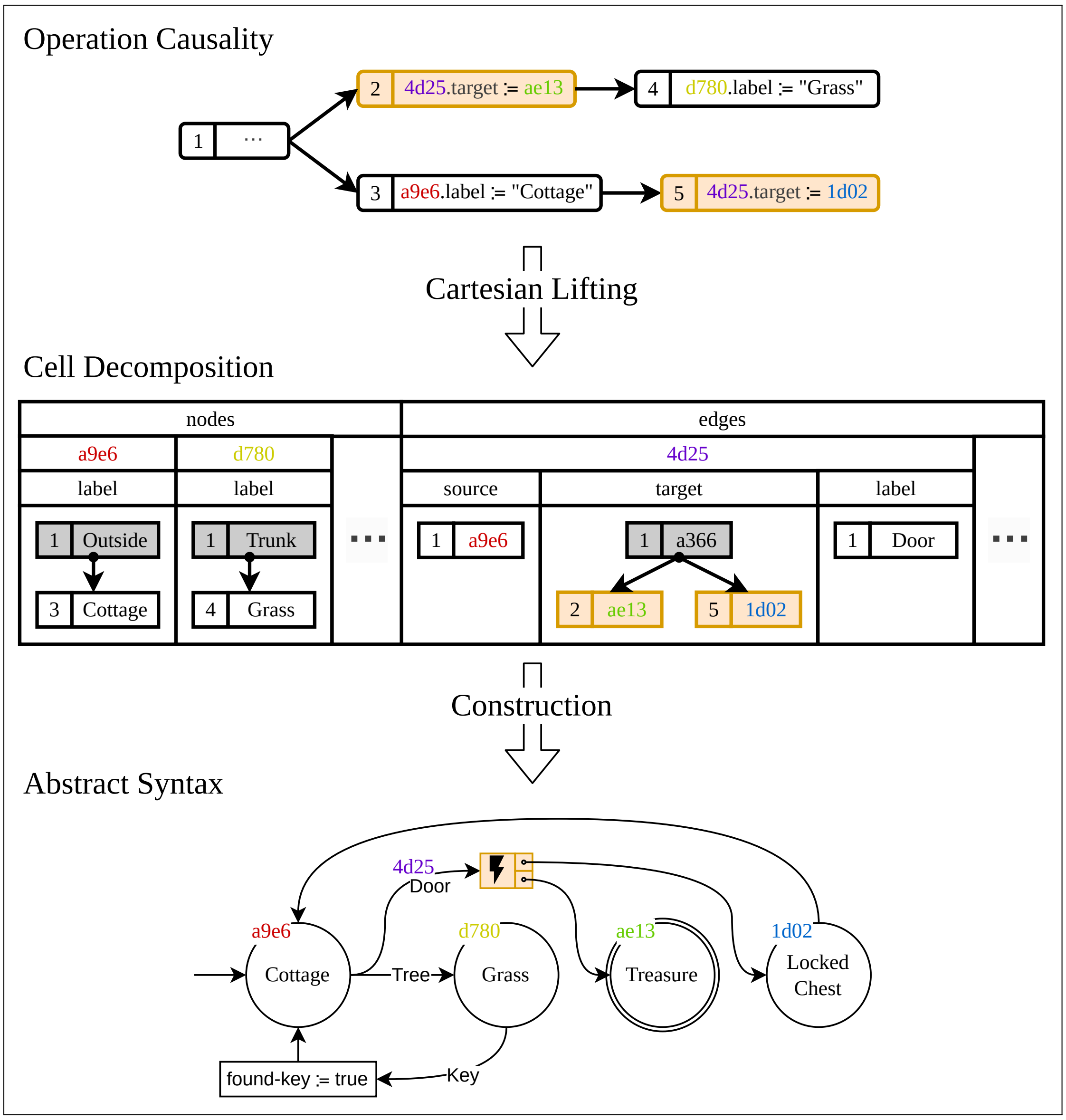
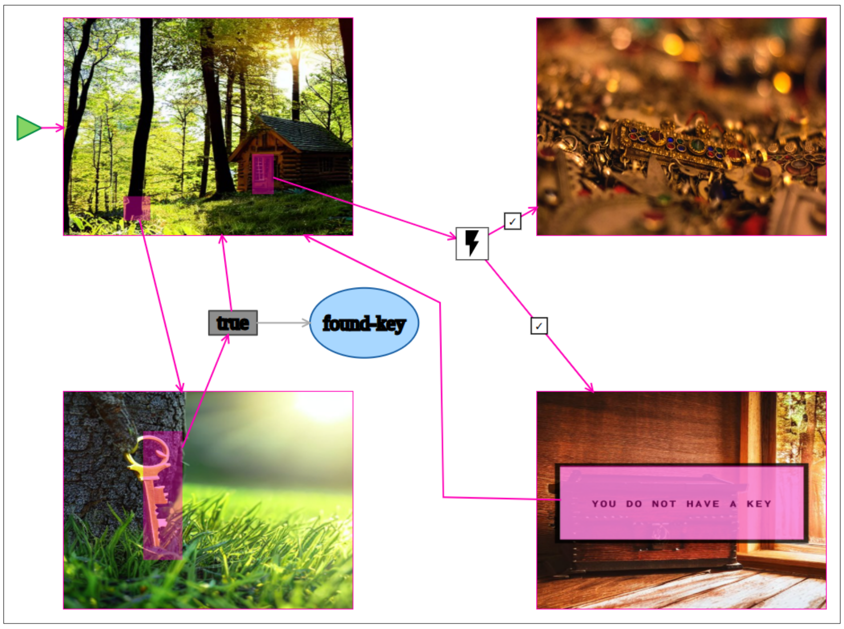


Figure 2: Lazy merging derives the current state of the abstract syntax from the causality of edit operations. After decomposing the global causality order across cells, the impact on abstract syntax elements becomes apparent.

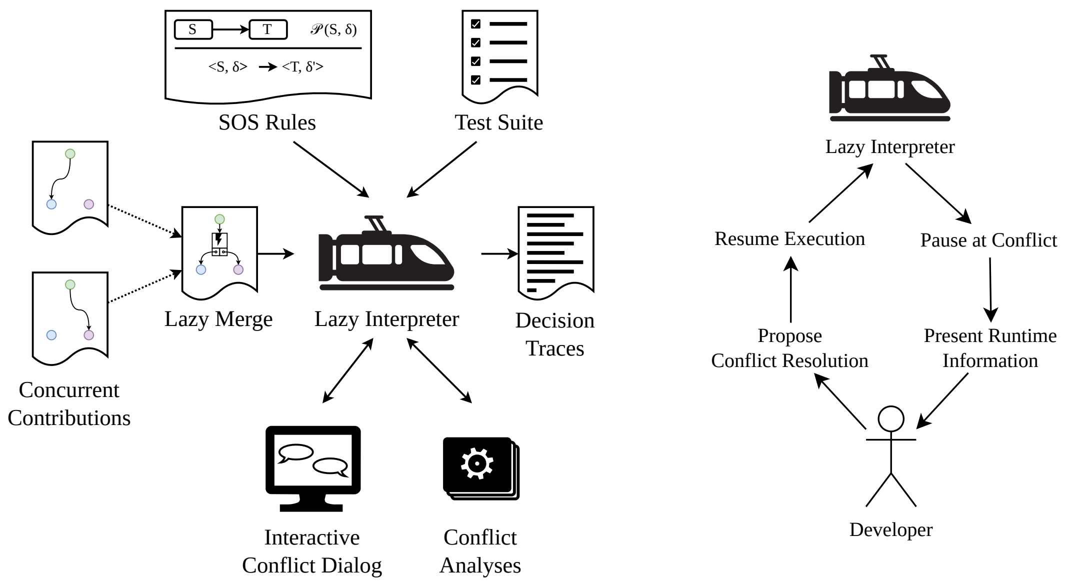


(a) Graphical surface syntax

```
start screen Cottage {
      image "cottage.jpg"
      rectangle Door at (100, 50)
      ⋮
}
Door leads to   Treasure
                Locked-Chest
screen Treasure { … }
screen Locked-Chest { … }
variable found-key
⋮
```

(b) Textual surface syntax

Figure 3: The conflicts in the abstract syntax are displayed with custom-tailored visualizations in different surface syntaxes for optimal comprehensibility.



(a) Overall architecture

(b) Interactive conflict dialog

Figure 4: When executing lazily merged programs, the lazy interpreter suspends the execution at conflict points, allowing developers and tools to analyze the situation and make decisions. The interpreter records detailed traces that include a log of all decisions.

on context-free grammars are commonplace. First, the cell segmentation must be applied to the context-free grammar. Syntax element productions are extended with a unique identifier and equipped with cells for all captured values and references. Then, a textual projectional editor is created for the new internal representation of the context-free grammar. This editor can be implemented, e.g., with JetBrains MPS [BCCP21], which provides a "text-like" structured editing model and readily available projectional editors for popular languages, such as Java. Rather than editing plaintext program representations with a text editor, developers work with a projectional textual representation that is enriched with conflict representations.

3 Lazy Interpretation

Current interpreters have no means to execute conflicted programs, because they are limited by plaintext program representations that cannot accommodate versioning concerns. Lazy merging offers a fresh approach with an internal program representation that incorporates conflict operators. This representation preserves the syntactic structure and embeds conflicting values into the source document. Furthermore, since the program representation is derived from the recorded edit history, there is a strong integration between history, syntax and conflicts. This program representation makes step-by-step interpretation straightforward: Conflict-free areas are just regular program syntax and conflicts provide possible choices and a link to their history. The actual surface syntax can still use a textual notation within a projectional conflict-aware editor, lazy interpretation is enabled by the new internal program representation.

Lazy interpretation transforms the entire software integration process. Developers no longer need to pick conflict resolutions based solely on their limited mental model of the program. Instead, they begin by executing the program and automated test suites to explore the situation. They can then debug the conflicted program interactively and investigate the runtime at conflict points. They can experiment with different choices for each conflict and immediately observe the impact on the runtime in the remaining execution. Developers can interactively build an understanding of conflicts and then resolve them incrementally whenever they feel confident.

Figure 4a shows the architecture of the lazy interpreter. Given input in the form of lazily merged programs, the interpreter either executes freely or follows automated test cases. When a conflict arises, the developer can decide how to proceed the execution on the fly (see Figure 4b). Analyses can support this decision-making process with suggestions and additional insights, or they can make decisions independently. The lazy interpreter records a detailed execution trace for each run. This trace includes the test result and all the decisions that have been made along the way, as well as a snapshot of the current runtime state for every decision. Each run adds to the growing collection of decision traces, which we can later use to construct an optimal resolution.

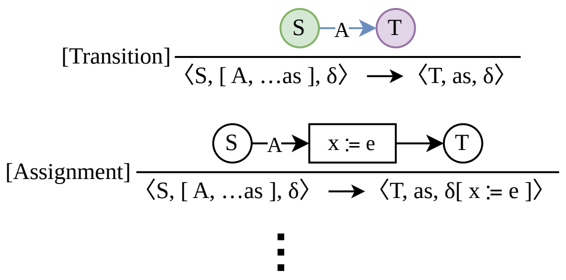
The lazy interpreter follows an SOS-based [Plo81] semantics specification in a graphical language inspired by our recent efforts on SOS-based graph transformation [TKS23]. In Figure 5a, each rule has a premise at the top and a conclusion at the bottom. The conclusion describes an execution step, where the state is a three-tuple consisting of the current screen, click actions, and the variable state. The premises pattern-match on the abstract syntax and may specify additional conditions. Figure 5b shows the interaction of the SOS rules with conflicts. When the pattern of the transition rule matches a conflict operator in the target node, the execution pauses and the developer is prompted to concretize the target node on the fly before the execution can resume.

Developers control and inspect the execution with debugging interfaces integrated into the graphical and textual projectional views as shown in Figure 6. When the execution is paused, the lazy interpreter displays the current runtime state along with the trace of decisions up to that point. Developers can then explore the program's dynamic behavior before deciding how to proceed the execution. They can either choose one of the available options from the conflict, or propose an entirely different resolution. In addition to resolving conflicts during execution, developers can also live-code further program specifications.

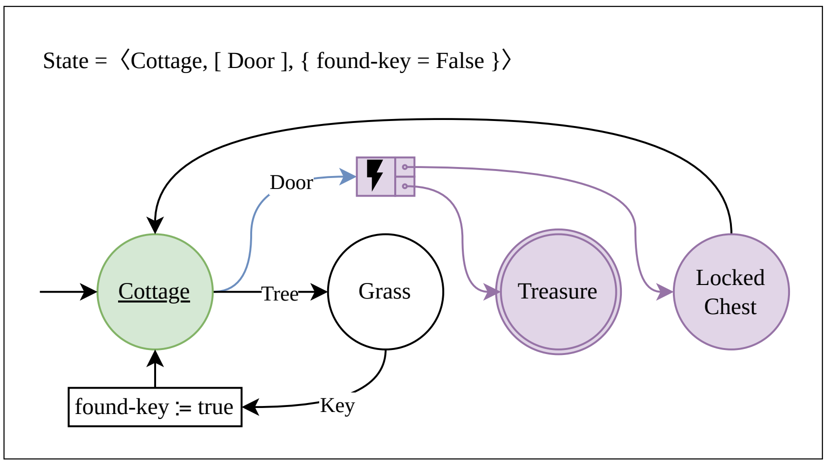
In addition to enabling interactive debugging and exploration, lazy interpretation allows for new automated analyses and synthesis strategies. The following sections describe how they utilize the lazy interpretation of test suites to find resolution proposals and synthesize conflict resolutions.

4 Test-By-Test Conflict Analysis

Test suites play a crucial role in guiding the interactive exploration and automating the search for conflict resolutions. They enable us to examine conflicts from the perspectives of different needs and usage scenarios, breaking down the difficult conflict resolution process. Since the tests provide expected output values for a wide range of different inputs, they can immediately

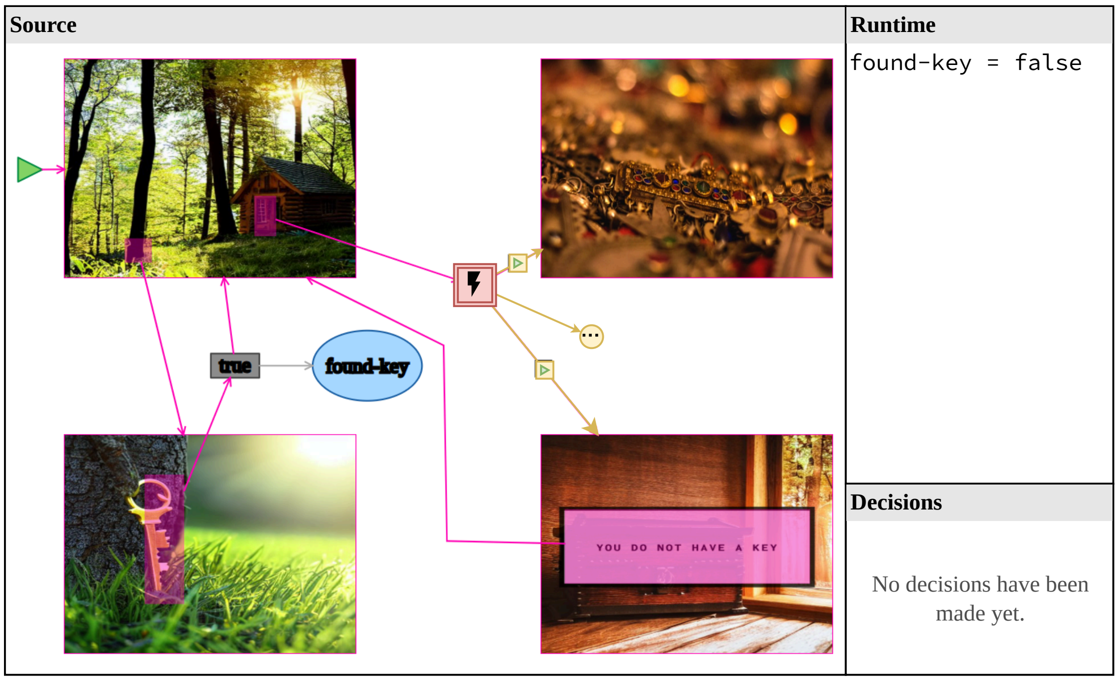

S
A
T
[Transition]
⟨S, [A, …as], δ⟩ ⟶ ⟨T, as, δ⟩
S
A
x := e
T
[Assignment]
⟨S, [A, …as], δ⟩ ⟶ ⟨T, as, δ[x := e]⟩

(a) SOS rules for point-and-click adventures



(b) Matching a conflicted syntax element

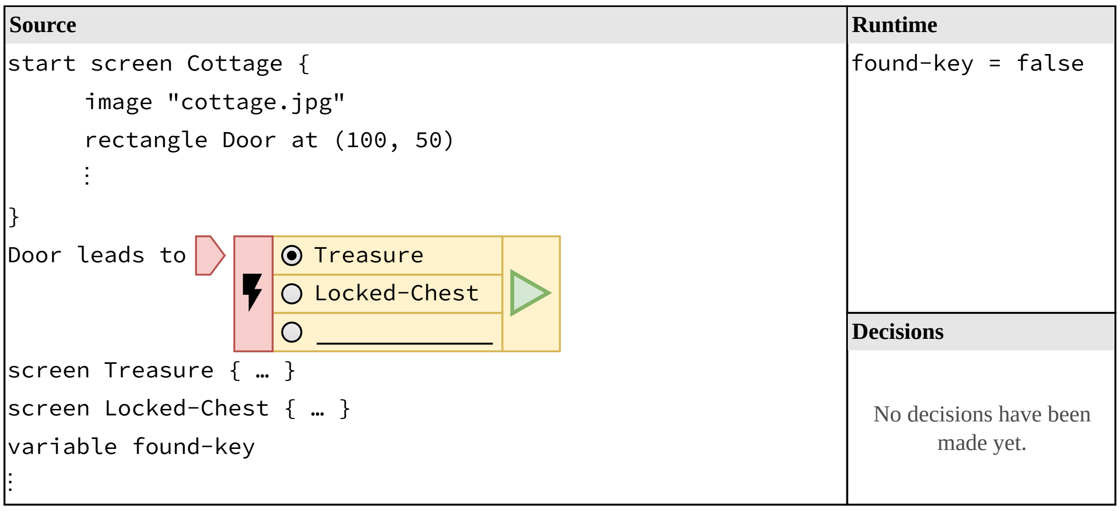
Figure 5: The lazy interpreter executes conflicted programs by pattern-matching SOS rules on their syntax. When conflicts are matched, the developer concretizes the syntax on the fly.



```
found-key = false
```

No decisions have been made yet.

(a) Graphical surface syntax



```
start screen Cottage {
      image "cottage.jpg"
      rectangle Door at (100, 50)
      ⋮
}
Door leads to   ● Treasure
                ○ Locked-Chest
                ○ ____________
screen Treasure { … }
screen Locked-Chest { … }
variable found-key
⋮
```

```
found-key = false
```

No decisions have been made yet.

(b) Textual surface syntax

Figure 6: The surface syntaxes extend their conflict visualization with debugging controls to let developers make ephemeral resolution decisions on the fly. The current program state is shown on the right.

This branch has conflicts that must be resolved

Conflicting files:

- `forest.story`

(a) GitHub conflict listing

Lazy Interpretation Test Report

Results

- 0 tests failed expectations
- 2 tests failed from conflicts
- 8 tests passed

Failures

`forest.story`

✗ No treasure without key

✗ Treasure with key

(b) Test report from lazy interpretation

Figure 7: Despite existing conflicts, the lazy interpreter can produce detailed test reports during continuous integration.

validate resolution proposals. If a test suite is insufficient, additional tests can be generated using search-based test generation [FA11], or recent GPT-based test generation approaches [DSL+24].

By simply executing test suites as far as possible, we can already gather valuable feedback on the runtime impact of conflicts. While current software forges such as GitHub can only locate conflicts syntactically, our approach precisely shows which test cases are actually broken by the conflicts. Thus, we can evaluate the severity of conflicts before integrating them. Figure 7 shows how GitHub only lists the syntactic locations of conflicts, whereas our approach provides the full test results. Furthermore, the lazy interpreter can determine which conflicts caused tests to fail, providing additional insight into the impact of conflicts. These results can enrich the reports of continuous integration systems [DHM+11] as well as the immediate feedback of conflict awareness systems [GS12, SCG12].

Figure 8 shows two tests for the point-and-click adventure game: Without a key, the player reaches a locked door; with a key, the player accesses the treasure. The lazy interpretation of these tests is a powerful tool for developers that restructures the entire integration process. Rather than resolving all conflicts at once, developers can now approach the search for resolutions test by test. Figure 9 illustrates how this resolution process might unfold by incrementally resolving the conflicts for more and more tests. First, a developer selects up a failing test and iterates until they find a suitable resolution proposal to make the test pass. At this stage, the developer does not need to consider any other part of the program and only searches for a proposal that resolves one specific scenario. Then, they reconcile the new resolution proposal with the consensus of the previous iterations. Here, the developer only has to consider how to integrate the new proposal into the existing consensus. Once a consensus is reached for all tests up to this point, they can move on to the next failing test case, until all tests are satisfied.

T$_1$ No treasure without key
Click
1. Door
arrives at
Locked-Chest

T$_2$ Treasure with key
Click
1. Tree
2. Key
3. Door
arrives at
Treasure

Figure 8: Tests describe the steps of story scenarios and validate the final destination.

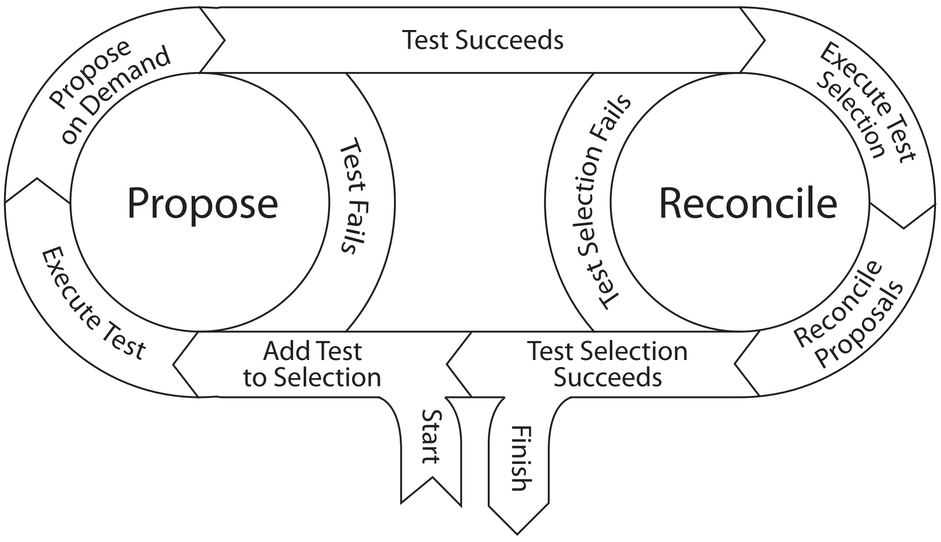


Figure 9: Through testing, developers can focus on individual scenarios and develop small and specific resolution proposals, before incrementally reconciling the proposals into a good overall resolution.

Through lazy interpretation, we can also analyze conflicting values in an attempt to select promising proposals and candidate combinations. To solve this challenging problem, we break it down test by test and focus the analyses on specific needs and requirements. We leverage historical analyses and the tight integration of version control with the abstract syntax to enable effective search strategies. By correlating the introduction of test cases with the assignment operations of conflicted values, we can find good resolution proposals that satisfy specific test cases. Additionally, we can also further analyze the arrangement of sequential chunks in the fine-grained history captured by lazy merging to discover logical connections between values from different conflicts.

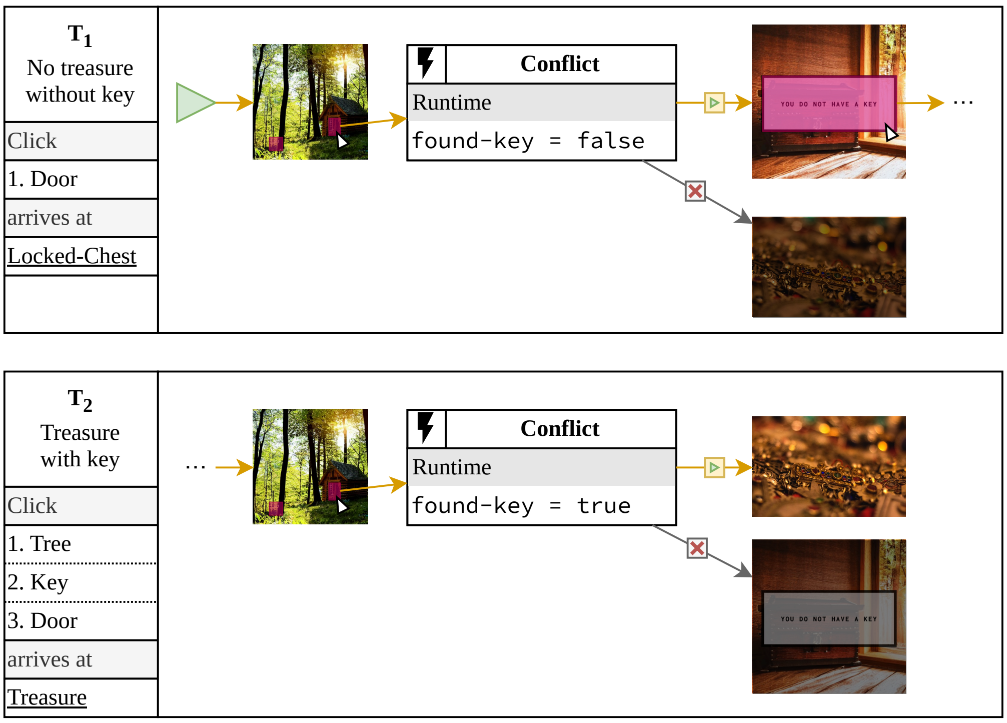
Sometimes, this test-by-test search for proposals already yields a resolution that satisfies all other test cases as well. At other times, neither proposal is sufficient for all test cases, as requirements or implementations contradict each other. In the following section we will discuss the synthesis of good overall resolutions when there is no single best proposal.

5 Conflict Resolution Synthesis

When test scenarios exhibit contradicting requirements, there will be no single resolution proposal that satisfies all tests. In this case, we attempt to reconcile the proposals by analyzing the runtime snapshots in the recorded decision traces to determine the kind of runtime state in which each proposal works. Once we understand how the runtime state influences the validity of the proposals, we can infer the logical conditions that differentiate between the scenarios at runtime. Then, we replace conflicts with branching points that conditionally select the appropriate resolution proposal based on the runtime state. Using this strategy, we combine several partially valid proposals into one good overall resolution.

Figure 10a shows the recorded traces, which illustrate how the resolution proposals differ between the two tests: In the first test, when the player clicks directly on the door, the developer selected the locked chest screen as the resolution to the conflict. In contrast, after the player had picked up the key in the second test, the developer selected the treasure screen in the same place. A comparison of the two proposals in Figure 10b shows that neither proposal is right for both tests, so the two choices must be reconciled. Examining the recorded runtime state in the two situations reveals that the `found-key` variable determines the next screen. If the key has been found, the treasure screen should appear; otherwise the locked chest screen should appear. Based on this observation, we can create a branching point in place of the conflict that selects the appropriate screen based on the value of the `found-key` variable. As seen in Figure 10c, inserting the branching point is a good overall resolution that satisfies both test cases. Figure 11 shows the synthesized resolution in both graphical and textual surface syntax. The inserted branching point is highlighted in green.

In general, automatically inferring an optimal logical condition remains an open problem. The runtime snapshots recorded in the traces provide valuable observations for machine learning approaches, such as genetic programming and decision tree learning. These approaches can directly generate logical conditions; however, it is still unclear which method yields good results in our use case. Test-driven lazy interpretation can also help by probing variables for their



(a) Decision traces for the two tests with different resolution proposals

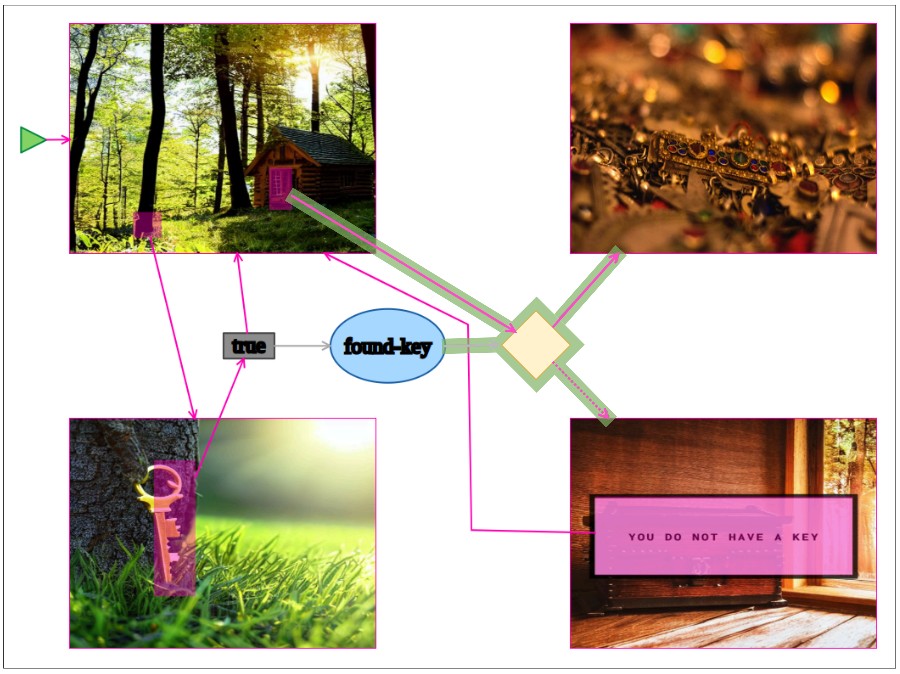
Proposal	Origin	Conflict 1		T_1	T_2
		Runtime	Choice		
P_1	T_1	`found-key = false`	`Locked-Chest`	✓	✗
P_2	T_2	`found-key = true`	`Treasure`	✗	✓

(b) Comparison of resolution proposals

Conflict 1	T_1	T_2
`if` `found-key` **`then`** `Treasure` **`else`** `Locked-Chest`	✓	✓

(c) Conflict resolution with generated branching point

Figure 10: Neither of the two resolution proposals shown in the decision traces is right for both tests. Only the insertion of a branching point based on the `found-key` variable produces a good overall resolution.



(a) Graphical surface syntax

```
start screen Cottage {
      image "cottage.jpg"
      rectangle Door at (100, 50)
      ⋮
}
Door leads to  Check-Key
screen Treasure { … }
screen Locked-Chest { … }
variable found-key
condition Check-Key {
      if found-key then
            Treasure
      else
            Locked-Chest
}
⋮
```

(b) Textual surface syntax

Figure 11: The generated branching point replaces the conflict for a good overall resolution.

relevance. If variables do not affect the validity of the selected proposals, they can be omitted from the condition to reduce the inference complexity.

6 Related Work

The idea of execution-first software development is not new, and it is perhaps most prominently embodied by the test-driven development paradigm. In this approach, the program specification is incrementally completed in a loop of writing a test, executing the incomplete program, and then extending the specification until the test passes [Bec03]. Bagherzadeh et al. adapted the execution-first concept into graphical modeling by executing partial state machine models. Live modeling enables developers to address lack of progress or reachability on the fly as the state machine model is executed. They also provide script-based automation and heuristics to support the decision-making process [BKJD22]. Recently, we have investigated test-first development in the domains of teaching and UI design [SPS+22]. While all of these works are similar to our proposed approach in spirit, they can only handle incomplete but consistent programs. They are unaware of versioning concerns and have no means to handle merge conflicts; therefore, they cannot support developers in the conflict resolution process.

Dong et al. also use test execution to resolve merge conflicts. They use tests to determine if one of two conflicting commits subsumes the other semantically, and then select it as the resolution [DSL+24]. Unlike our approach, theirs only handles merge conflicts where one of the commits is actually a satisfactory solution for both participants. If changes from both commits need to be combined or if new code is required, they cannot find a resolution. Thus, their approach is unsuitable for many merge scenarios. Our approach goes one step further and generates branching points to combine competing changes into a conflict resolution that works for both sides.

Nguyen et al. use a test-driven approach with variability-aware execution [NKN14] to efficiently detect semantic conflicts between *n* branches. They also introduce a new program representation that includes all concurrent changes to the program, as well as switches for all patches that enable or disable the corresponding changes. Their approach can efficiently identify semantic conflicts and determine which changes caused them [NND+15]. Although they use tests similarly, they solve a different problem. Our internal program representation includes only the conflicting changes of actual commits, not all concurrent changes. While Nguyen et al. aim to detect conflicts, we aim to resolve them. Nevertheless, their approach can complement our work nicely: The semantic conflicts that their variability-aware execution uncovers can be recorded in our conflict-tolerant program representation and then resolved with the same toolkit that we already use for syntactic conflicts.

Since 2022, several learning-based approaches for conflict resolution synthesis have been proposed. They are based on LLMs (prompting GPT-3 [ZMK+22] and ChatGPT [SYPZ23]), encode-decoder architectures (MergeGen [DZS+23], DeepMerge [DMS+23]), neural transformers (MergeBERT [SFG+22]), reinforcement learning [SZS22], and various classification algorithms [ESOM23]. In general, these approaches learn directly from historical data to either classify resolution strategies or generate complete resolutions. While these learning-based approaches offer decent performance, their broad use of machine learning renders the decision-making process difficult to understand. Our structured approach creates an understanding of conflicts and generates valuable data on the runtime behavior of conflicted programs, which transparently supports the decision-making process. We aim to use machine learning only for specific, limited tasks, such as synthesizing logical conditions to combine competing resolution proposals.

7 Conclusion

Our test-driven conflict resolution paradigm turns the software integration process on its head, offering exciting opportunities and overcoming existing limitations. We presented a lazy interpreter that enables developers to begin the software integration process with an interactive, test-based exploration of the runtime behavior. The interpreter empowers developers to first build a deep understanding of conflicts and investigate their interconnection in immersive experimental debugging sessions. They can make resolution decisions after careful deliberation, whenever they feel ready. Rather than resolving all conflicts upfront, developers can now incrementally resolve them one test at a time.

Furthermore, our approach paves the way for test-based conflict resolution synthesis: Test-based exploration yields a substantial dataset of runtime data about conflicts that can be mined for semantic insights. The program representation, which integrates syntax, history, and conflicts, allows for comprehensive historical analyses, while the test-based feedback validates hypotheses. Together, these elements support the synthesis of conflict resolutions that effectively integrate competing concerns.

Once our approach is fully implemented, we will evaluate two key research questions: First, we will evaluate the impact of exploration opportunities and lazy decision-making on integration work and collaborative conflict resolution. We will examine how test-driven conflict resolution can improve the understanding of conflicts and support developers in the interactive search for resolutions. We will examine this question in a workshop evaluation and compare our results to those of other software integration workshops that did not offer test-driven exploration [WLS+12]. Second, we will investigate how to infer branching conditions to reconcile partial proposals and fully realize test-driven resolution synthesis. The key challenge will be finding a good inference technique that can leverage our runtime analysis data and generate good differentiating conditions. We will compare the performance of our conflict resolution synthesis with the existing test-based search strategies and recent learning-based approaches that have been presented in Section 6. To make the results comparable, we will transfer their Git-based evaluation corpora into our internal program representation to simulate the use of lazy merging in the given scenarios. Precision and recall of conflict resolutions will be the key metrics of this evaluation.

Bibliography

[BBK+22] A. Bainczyk, D. Busch, M. Krumrey, D. S. Mitwalli, J. Schürmann, J. Tagoukeng Dongmo, B. Steffen. CINCO Cloud: A Holistic Approach for Web-Based Language-Driven Engineering. In Margaria and Steffen (eds.), *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering*.

Pp. 407–425. Springer Nature Switzerland, Cham, 2022.
doi:10.1007/978-3-031-19756-7_23

[BBR$^+$12] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, P. Devanbu. Cohesive and Isolated Development with Branches. In Lara and Zisman (eds.), *Fundamental Approaches to Software Engineering*. Pp. 316–331. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
doi:10.1007/978-3-642-28872-2_22

[BCCP21] A. Bucchiarone, A. Cicchetti, F. Ciccozzi, A. Pierantonio. *Domain-Specific Languages in Practice: with JetBrains MPS*. Springer International Publishing, 2021.
doi:10.1007/978-3-030-73758-0

[Bec03] K. Beck. *Test-driven Development: By Example*. Addison-Wesley signature series. Addison-Wesley, 2003.

[BKJD22] M. Bagherzadeh, N. Kahani, K. Jahed, J. Dingel. Execution of Partial State Machine Models. *IEEE Transactions on Software Engineering* 48(3):951–972, March 2022.
doi:10.1109/TSE.2020.3008850

[DHM$^+$11] P. Debois, J. Humble, J. Molesky, E. Shamow, L. Fitzpatrick, M. Dillon, B. Phifer, D. DeGrandis. Devops: A Software Revolution in the Making? *Journal of Information Technology Management* 24(8):3–39, 2011.

[DMS$^+$23] E. Dinella, T. Mytkowicz, A. Svyatkovskiy, C. Bird, M. Naik, S. Lahiri. DeepMerge: Learning to Merge Programs. *IEEE Transactions on Software Engineering* 49(4):1599–1614, April 2023.
doi:10.1109/TSE.2022.3183955

[DSL$^+$24] J. Dong, J. Sun, Y. Lin, Y. Zhang, M. Ma, J. S. Dong, D. Hao. Revisiting the Conflict-Resolving Problem from a Semantic Perspective. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’24, p. 141–152. Association for Computing Machinery, New York, NY, USA, 2024.
doi:10.1145/3691620.3694993

[DZS$^+$23] J. Dong, Q. Zhu, Z. Sun, Y. Lou, D. Hao. Merge Conflict Resolution: Classification or Generation? In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Pp. 1652–1663. Sep. 2023.
doi:10.1109/ASE56229.2023.00155

[ESOM23] P. Elias, H. de Souza Campos, E. Ogasawara, L. G. P. Murta. Towards accurate recommendations of merge conflicts resolution strategies. *Information and Software Technology* 164:107332, 2023.
doi:10.1016/j.infsof.2023.107332
https://www.sciencedirect.com/science/article/pii/S0950584923001878

[FA11] G. Fraser, A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11, p. 416–419. Association for Computing Machinery, New York, NY, USA, 2011. doi:10.1145/2025113.2025179

[GS12] M. L. Guimarães, A. R. Silva. Improving early detection of software merge conflicts. In *2012 34th International Conference on Software Engineering (ICSE)*. Pp. 342–352. June 2012. doi:10.1109/ICSE.2012.6227180

[Men02] T. Mens. A State-of-the-Art Survey on Software Merging. *Software Engineering, IEEE Transactions on* 28:449–462, June 2002. doi:10.1109/TSE.2002.1000449

[NKN14] H. V. Nguyen, C. Kästner, T. N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014, p. 907–918. Association for Computing Machinery, New York, NY, USA, 2014. doi:10.1145/2568225.2568300

[NND$^+$15] H. V. Nguyen, M. H. Nguyen, S. C. Dang, C. Kästner, T. N. Nguyen. Detecting semantic merge conflicts with variability-aware execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015, p. 926–929. Association for Computing Machinery, New York, NY, USA, 2015. doi:10.1145/2786805.2803208

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical report, University of Aarhus, 1981. DAIMI FN-19.

[PSW11] S. Phillips, J. Sillito, R. Walker. Branching and Merging: An Investigation into Current Version Control Practices. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, may 2011. doi:10.1145/1984642.1984645

[SCG12] I. Steinmacher, A. P. Chaves, M. A. Gerosa. Awareness Support in Distributed Software Development: A Systematic Review and Mapping of the Literature. *Computer Supported Cooperative Work (CSCW)* 22(2-3):113–158, May 2012. doi:10.1007/s10606-012-9164-4

[SFG$^+$22] A. Svyatkovskiy, S. Fakhoury, N. Ghorbani, T. Mytkowicz, E. Dinella, C. Bird, J. Jang, N. Sundaresan, S. K. Lahiri. Program merge conflict resolution via neural transformers. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022, p. 822–833. Association for Computing Machinery, New York, NY, USA, 2022. doi:10.1145/3540250.3549163

[SPS+22] S. Smyth, J. Petzold, J. Schürmann, F. Karbus, T. Margaria, R. von Hanxleden, B. Steffen. Executable Documentation: Test-First in Action. Proc. ISoLA 2022, 2022. [in this volume].

[SS23] J. Schürmann, B. Steffen. Lazy Merging: From a Potential of Universes to a Universe of Potentials. *Electronic Communications of the EASST* Volume 82: 11th International Symposium on Leveraging Applications of Formal Methods, 2023.
doi:10.14279/tuj.eceasst.82.1226

[SYPZ23] C. Shen, W. Yang, M. Pan, Y. Zhou. Git Merge Conflict Resolution Leveraging Strategy Classification and LLM. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. Pp. 228–239. Oct 2023.
doi:10.1109/QRS60937.2023.00031

[SZS22] M. Sharbaf, B. Zamani, G. Sunyé. Automatic resolution of model merging conflicts using quality-based reinforcement learning. *Journal of Computer Languages* 71:101123, 2022.
doi:10.1016/j.cola.2022.101123
https://www.sciencedirect.com/science/article/pii/S2590118422000260

[TKS23] S. Teumert, M. Krause, B. Steffen. SOS-Supported Graph Transformation. *Electronic Communications of the EASST* Volume 82: 11th International Symposium on Leveraging Applications of Formal Methods, 2023.
doi:10.14279/tuj.eceasst.82.1220

[WLS+12] K. Wieland, P. Langer, M. Seidl, M. Wimmer, G. Kappel. Turning Conflicts into Collaboration. *Computer Supported Cooperative Work (CSCW)* 22(2-3):181–240, Sept. 2012.
doi:10.1007/s10606-012-9172-4

[ZMK+22] J. Zhang, T. Mytkowicz, M. Kaufman, R. Piskac, S. K. Lahiri. Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022, p. 77–88. Association for Computing Machinery, New York, NY, USA, 2022.
doi:10.1145/3533767.3534396