

Formal Verification of Repository Synchronization and Merge Consistency in a Distributed Version Control System

Bekzat Ongdassynov and Sultan Kasenov

Nazarbayev University, School of Engineering & Digital Sciences, Astana, Kazakhstan

Authors' email: bekzat.ongdassynov@nu.edu.kz and sultan.kasenov@nu.edu.kz

Abstract—Distributed version control systems like Git manage complex operations across multiple repositories with concurrent updates and asynchronous communication. This paper presents a formal model of a Git-like distributed version control system using SPIN model checker and Promela specification language. We built a base model that verifies structural properties of commits, branches, and merges across two repositories. We then extended the model with content-aware conflict detection by tracking file modifications. Exhaustive state-space exploration verified all correctness properties in both models. The base model explored 62 million states, and the extended model verified 192 million states with conflict detection. All properties held without errors. This work shows how formal verification techniques can be applied to verify distributed version control operations.

I. INTRODUCTION

Distributed version control systems (DVCS) have become central to modern software development. They support collaboration by recording and propagating changes across multiple repositories. Systems such as Git enable developers to work independently, maintain complete local histories, and synchronize contributions without relying on a single authoritative server[1].

Version control environments must manage non-deterministic development activity across many distributed clients. Multiple developers can modify files simultaneously. Branches diverge and later merge. Network delays cause asynchronous message delivery. This creates many possible execution orderings that the system must handle correctly[2]. These interleavings are especially problematic at merge time, when independently evolved branches need to be integrated.

Merge conflicts occur when two developers modify overlapping sections of a single file. Such a scenario requires manual intervention. Semantic inconsistencies can also occur even when individual changes are correct on their own[3]. For example, one branch might rename a function while another branch continues to use the old name. Modern version control systems use three-way merge procedures with a base version and two diverging histories[4].

Model checking provides a way to verify these systems formally. SPIN model checker can exhaustively explore all possible execution interleavings[5]. This provides stronger guarantees than testing alone. We can verify that important properties hold in all scenarios.

This paper applies formal verification to distributed version control operations. We build a model in Promela and verify it using SPIN. The work consists of:

- A formal Promela model of Git-like operations including commits, branches, merges, and push
- An extension with file-level modification tracking and conflict detection
- Verification results showing all properties hold across millions of states
- Analysis of parameter tuning to balance verification thoroughness with computational tractability

The remainder of this paper is organized as follows. Section III describes the system model and formalization approach. Section IV presents the content-aware extension with conflict detection. Section V explains the verification methodology. Section VI presents experimental results. Section VII discusses implications and limitations. Section VIII outlines possible extensions. Finally, Section IX concludes the paper.

II. RELATED WORK

The SPIN model checker was originally developed for verifying network protocols and concurrent systems, not high-level distributed version control workflows. However, as collaborative software development has grown more complex, researchers have increasingly turned to automated model checking to reason about synchronization logic. Existing work shows that applying formal methods to DVCS is not only feasible, but also provides strong mathematical guarantees for a range of software engineering tasks.

For general merge problems in version control systems, Vale et al. [6] developed an analysis framework that combines repository mining with developer survey data. Their results characterize the difficulty of resolving merge conflicts and emphasize the need for more advanced support tools in complex scenarios. Cavalcanti et al. [7] explored a semi-structured merge protocol for Java repositories and showed that it reduces false positives and prevents build failures even when traditional textual merging fails. These works highlight the importance of direct state-space verification tools such as SPIN and Promela for modeling commit graphs and validating DVCS workflows.

Several studies have investigated how formal verification can ensure consistency in distributed systems. Gomes et al. [8] proposed a verification framework that combines network

failure models with strong eventual consistency (SEC) proofs to guarantee convergence in collaborative settings. Sun et al. [9] carried out an experimental comparison of Operational Transformation (OT) and Conflict-free Replicated Data Types (CRDT) for consistency maintenance in real-time collaborative editors. They found that OT better preserves user intention than CRDT in low-latency environments and introduced a general transformation framework that unifies these concurrency control approaches. Blau et al. [10] presented a Delta-CRDT model specified in Isabelle/HOL, reducing both state-transfer overhead and verification complexity. By proving the correctness of delta-based updates, they demonstrated efficient synchronization for wide area distributed systems.

In this work we build a SPIN-based formal model of a Git-like DVCS that captures commit-graph structure and file-level conflict detection.

III. SYSTEM MODEL

A. DVCS Operations Overview

A distributed version control system manages code changes across multiple repositories. Each repository maintains a complete history of commits. Commits form a directed acyclic graph where each commit points to one or more parent commits. Repositories have multiple branches that can diverge and later merge. Each branch has a HEAD pointer indicating the latest commit. Developers perform operations locally and then synchronize with remote repositories.

In our model, we focus on five main operations. A *commit* creates a new commit on the current branch. A *push* sends local commits to a remote repository asynchronously. A *merge* combines two branches by creating a commit with two parents. A *checkout* switches the active branch. Finally, a *fast-forward* updates a branch pointer when the target history is a strict extension of the current one. These operations execute concurrently across repositories. Network delays cause asynchronous message delivery. This creates non-deterministic execution orders that must all preserve consistency.

Table I shows a concrete execution trace illustrating how these operations interact. Repo 0 and Repo 1 independently create commits. Repo 0 pushes to Repo 1 while Repo 1 checks out a feature branch and commits there. The network propagates the push asynchronously. When Repo 0 later tries to merge the feature branch, it cannot proceed because commit C2 has not yet been replicated to Repo 0. This demonstrates how asynchronous communication creates complex dependencies that our model must handle correctly.

T	Repo 0	Repo 1	Network
1	commit C1 on main; heads[0][main]=1		
2		commit C1 on main; heads[1][main]=1	
3	push C1 to Repo1		[C1 msg]
4		checkout feature; active_branch[1]=1	
5		commit C2 on feat; heads[1][feat]=2	
6		fast-forward OK (C1 already there)	deliver
7	commit C3 on main; heads[0][main]=3		
8	merge feat. into main (impossible: no C2)		

TABLE I: Example DVCS execution trace showing concurrent operations and asynchronous synchronization. Final state: Repo 0 has {C1,C3} with main>C3; Repo 1 has {C1,C2} with main>C1, feature>C2.

B. Formal Problem Statement

We aim to verify that a DVCS maintains consistency under concurrent operations. Essentially, we check that the commit graph remains acyclic at all times, that branch pointers always reference commits that exist in the repository, that merge commits have exactly two parent commits, and that the repository state remains consistent under asynchronous operations.

We model the system as a concurrent batch of multiple repository processes. Each process makes non-deterministic choices and the model checker exhaustively explores all possible interleavings of these choices within the given rules.

The scope of the model with the main assumptions and abstractions is summarized in Table II. These bounds keep state space manageable. The model still captures all essential concurrent behaviors of a DVCS. We verified that 3 commits enough to test complex branching and merging scenarios.

Aspect	Setting / assumption
Repositories	$N_{\text{REPO}} = 2$
Branches per rep	$N_{\text{BRANCHES}} = 2$
Max n of commits	$\text{MAX_COMMITTS} = 3$
Operations per repo	$\text{MAX_STEPS} = 3$
Network model	Asynchronous, 4-message buffer
Fast-forward detection	Enabled during push operations
Message delivery	All messages are eventually delivered.
Operation granularity	Operations are atomic.
Conflict detection	Conflicts detected at file level.
Commit identifiers	Commits numbered sequentially.
Commit hashes	SHA-1 hashes abstracted as IDs 1, 2, 3.
File contents	File contents abstracted as file IDs 1, 2.
Advanced operations	Operations such as rebase are excluded.

TABLE II: Model scope, assumptions, and abstractions.

Commit structure:

```
typedef Commit {
  id: 2
  parent1: 1 // points to commit 1
  parent2: 0 // no second parent
  exists: true
  is_merge: false
}
```

Example commit graph:

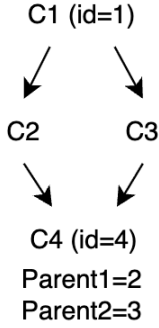


Fig. 1: Commit structure and example DAG

C. Commit Graph Structure

The commit graph is a directed acyclic graph. Each commit has zero, one, or two parents. Regular commits have one parent. Merge commits have two parents. The initial commit has no parents.

We ensure acyclicity by construction. Commits are numbered sequentially starting from 1. A commit can only reference parents with lower IDs. This topological ordering makes cycles impossible. Zero is reserved as a value that means "no commit" or "no parent."

Figure 1 shows a typical scenario. Commit C1 is the initial commit. C2 and C3 branch from C1. C4 merges C2 and C3 back together. Notice that parent IDs are always less than the commit ID, which prevents cycles.

D. Repository State

Each repository maintains local state:

heads[repo][branch]: Each branch has a HEAD pointer. This points to the commit at the tip of that branch. A value of 0 means the branch is empty.

active_branch[repo]: Each repository has one checked-out branch at a time. New commits go on the active branch.

repo_has[repo][commit]: This tracks which commits exist in each repository. A repository must have a commit before it can reference it in operations.

Repository 0 state:

```
heads[0][main]=2 > HEAD of main > C2
heads[0][feature]=3 > HEAD of feat > C3
active_branch[0]=0 > currently main
```

```
repo_has[0][1]=1 (has C1)
repo_has[0][2]=1 (has C2)
repo_has[0][3]=0 (no C3)
```

Visual representation (* means active):

```
Repo 0:  main* -> [C2]
         feature -> [C3]
         Has: {C1, C2}
```

Fig. 2: Repository state example

These arrays are flattened into one-dimensional arrays. We use macros HEAD(r,b) and HAS(r,c) to access them like 2D arrays. This is necessary because Promela does not support true multidimensional arrays.

Variable	Type	Description
commits[]	Commit[]	Global commit pool
heads[r][b]	byte	Branch head pointers
active_branch[r]	byte	Active branch per repo
repo_has[r][c]	bit	Commit replication state

TABLE III: State variables

E. Operations as State Transitions

We model five main operations:

local_commit(repo):

- 1) Get parent = HEAD of active branch
- 2) Create new commit: id = next_commit++
- 3) Set commit.parent1 = parent, commit.parent2 = 0
- 4) Update HEAD of active branch to new commit
- 5) Mark repo_has[repo][new commit] = true

push(src, dst):

- 1) Get commit_id = HEAD of active branch in src
- 2) Send message (src, dst, branch, commit_id) to network channel
- 3) Network delivers asynchronously (handled by Network process)

Network process handles delivery:

- 1) Receive (src, dst, branch, commit_id) from channel
- 2) Mark commit_id and all ancestors as present in dst repository
- 3) Check if fast-forward possible: commit descends from dst HEAD
- 4) If yes, update dst HEAD to commit_id
- 5) If no, dst must merge manually

merge_commits(repo, c1, c2):

- 1) Verify repo_has[repo][c1] = true and repo_has[repo][c2] = true

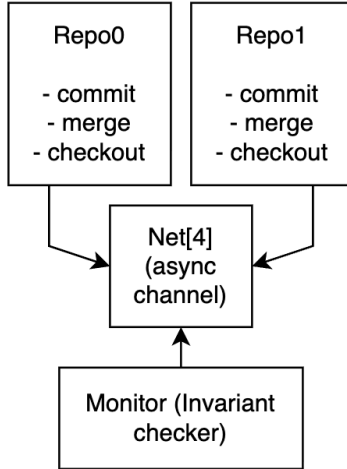
- 2) Create merge commit: `id = next_commit++`
- 3) Set `commit.parent1 = c1`, `commit.parent2 = c2`, `commit.is_merge = true`
- 4) Update HEAD of active branch to merge commit
- 5) Mark `repo_has[repo][merge commit] = true`

checkout_branch(repo, branch_id):

- 1) Set `active_branch[repo] = branch_id`

Each repository process non-deterministically chooses operations. SPIN explores all possible interleavings of these choices across both repositories and the network.

Process architecture:



Message flow example:

Repo0 --[push C2 to Repo1]--> Network
 Network --[deliver C2]--> Repo1
 (marks Repo1.has[C2] = true)
 (updates Repo1.heads if fast-forward)

Fig. 3: System architecture and message flow

IV. CONTENT-AWARE EXTENSION

A. Motivation for Content Modeling

The base model verifies structural properties of the commit graph only. In contrast, real version control systems operate on file contents: merge conflicts arise when two branches modify the same file. To be able to resolve such conflicts, the model must track (an abstraction of) file level changes.

Encoding full file contents would immediately explode the state space, therefore we adopt a very minimalist extension of it - each commit records *which* file it modifies, but not *how*. This is sufficient to verify that merge operations correctly catch conflicts whenever both parents modify the same file.

B. File Modification Tracking

We add file and conflict metadata to the original `Commit` record and the extended structure is shown in Figure 4.

In the extended model, we represent two abstract files (`NFILES = 2`). When a new commit is created, the repository process non-deterministically chooses either file1 or file2

```

typedef Commit {
  byte id;
  byte parent1;
  byte parent2;
  bit exists;
  bit is_merge;
  byte modified_file; // 0 = none, 1 or 2 = file ID
  bit has_conflict; // Set during merge
}
  
```

Fig. 4: Extended Commit structure

Linear commit history with file modifications:
 C1 (file: 1) -> C2 (file: 2) -> C3 (file: 1)

Merge scenarios:

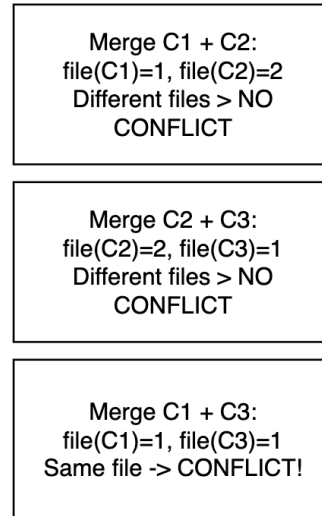


Fig. 5: File modification and merge outcomes

as the modified file. The SPIN model checker explores both choices, covering all possible patterns of file modifications without explicitly enumerating any of their contents.

Figure 5 shows how file tracking affects merge conflict detection. C1 and C3 both modify file1, C2 modifies file2. Merging commits that modify different files succeed without conflict. Merging commits that modify the same file triggers conflict detection.

C. Conflict Detection Algorithm

We characterize merge conflicts only in terms of the parents' 'modified file' metadata. For parent commits c_1 and c_2 , a merge is conflicting if and only if both parents modify a

Branching scenario with conflict:

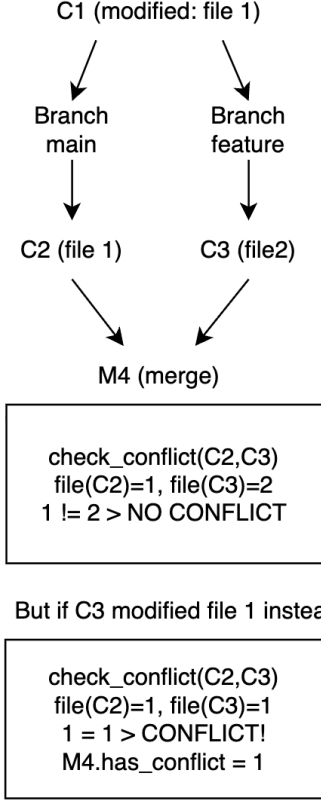


Fig. 6: Conflict detection in merge commits

(nonzero) file and the corresponding file identifiers coincide:

$$\text{conflict}(c_1, c_2) \equiv (\text{mf}(c_1) > 0) \wedge (\text{mf}(c_2) > 0) \wedge (\text{mf}(c_1) = \text{mf}(c_2)) \quad (1)$$

where $\text{mf}(c)$ denotes the *modified_file* field of commit c .

The Promela implementation of (1) is encapsulated in an inline procedure:

Listing 1: Promela inline for conflict detection

```

inline check_conflict(c1, c2, result) {
  if
  :: (commits[c1].modified_file > 0) &&
    (commits[c2].modified_file > 0) &&
    (commits[c1].modified_file ==
     commits[c2].modified_file) ->
    result = 1
  :: else ->
    result = 0
  fi
}

```

During merge operations, we call this function with both parent commits. The result is stored in the merge commit's `has_conflict` field.

Parameter	Base	Ext.	Justification
MAX_COMMITS	3	3	Tests branching and merging, critical for graph structure
MAX_STEPS	3	3	Enough operations to test async scenarios
NFILES	N/A	2	Minimal for conflict detection semantics
Monitor	80	50	Reduced to save memory, violations appear early anyway

TABLE IV: Final model configuration

Figure 6 illustrates conflict detection. When creating merge commit M4, we check if both parents modify the same file. If they do, we set `M4.has_conflict = 1`.

D. Parameterization and Bounds

We initially attempted larger bounds. In fact, state space grows exponentially with parameters. We went through several iterations to find workable parameters.

1) *Parameter Tuning Process*: Our first attempt used `MAX_COMMITS=5` and `MAX_STEPS=5`. This seemed reasonable for testing deeper commit graphs. However, the verifier was killed after exceeding available memory. The state space exceeded 500 million states.

We faced a choice: reduce `MAX_COMMITS` or reduce `MAX_STEPS`. `MAX_COMMITS` affects the commit graph depth. More commits allow testing longer chains of merges. `MAX_STEPS` affects operation sequences. More steps let repositories perform more operations. Even though we could see that the verifications would hold with greater parameter values and consequently more interleavings, partial runs and killed processes provide no concrete guarantees.

We decided `MAX_COMMITS` was more important for verification meaning. The structure of the commit graph is central to version control correctness, but further experimentation resulted in reduction in both. We reduced `MAX_COMMITS` to 4 and `MAX_STEPS` to 4.

This still produced too many states. We further reduced to `MAX_COMMITS=3` and `MAX_STEPS=3`. This completed successfully.

For `NFILES`, we initially included 0 to represent "no file modified." This added a choice but little verification value. We removed the zero option, so commits always modify files 1 or 2. This reduced the state space while preserving conflict detection semantics.

The Monitor bound parameter controls how many iterations the monitoring loop performs. We started at 150, then reduced to 100, 80, and finally 50. Lower values reduce memory just slightly but risk missing late-occurring invariant violations. We found 50 sufficient since violations typically appear early if they exist.

Try	MAX_COMMITS	MAX_STEPS	Result
1	5	5	Killed (>500M states)
2	4	4	Still too large
3	3	3	Success (192M states)

TABLE V: Parameter tuning iterations

The final configuration balances thoroughness with tractability. Three commits suffice to test branching, merging, and conflict scenarios. Two files provide meaningful conflict detection. The verification completes in under 7 minutes with 30GB memory.

The state space growth was dramatic: MAX_COMMITS=5 produced over 500 million states and was killed; MAX_COMMITS=4 generated approximately 350 million states which was too slow; MAX_COMMITS=3 successfully completed with 192 million states.

When deciding which parameter to reduce, we considered two options. Option 1 was to reduce MAX_COMMITS, which tests commit graph structure and is core to DVCS correctness - this is critical and should be kept as high as viable. Option 2 was to reduce MAX_STEPS, which tests operation sequences and is less critical than graph depth - making it the better candidate to reduce first. Based on this analysis, we settled on the final configuration: MAX_COMMITS=3 and MAX_STEPS=3.

E. Additional Properties

The extended model adds two new LTL properties.

a) *Conflict only in merges.*: If a commit has a conflict, then it must be a merge commit:

$$\Box \forall c : \text{conflict}(c) \Rightarrow \text{merge}(c). \quad (2)$$

This ensures that conflicts only arise during merge operations; ordinary (commits with a single parent) can never be marked as conflicting.

b) *Valid file modifications.*: If a commit exists, its modified file identifier must be within bounds:

$$\Box \forall c : \text{exists}(c) \Rightarrow \text{modifiedFile}(c) \leq N_{\text{files}}. \quad (3)$$

This property verifies that file identifiers never exceed the allowed range (1 or 2 in our model).

V. VERIFICATION METHODOLOGY

A. Implementation Details

We implement both models in Promela, the specification language for SPIN model checker. The base model is 402 lines. The extended model with conflicts is 483 lines.

The system consists of four concurrent processes:

- **Repo(0)**: First repository process
- **Repo(1)**: Second repository process
- **Network()**: Asynchronous message delivery
- **Monitor()**: Continuous invariant checking

Each Repo process loops up to MAX_STEPS times. At each step, it non-deterministically chooses an operation: commit,

push, merge, or checkout. SPIN explores all possible choices and interleavings.

The Network process uses an asynchronous channel with 4-message buffer capacity. This models realistic network delays. Messages can arrive in any order. SPIN verifies all orderings.

The Monitor process continuously checks invariants using assertions. It also provides LTL property verification. The monitor runs concurrently with other processes to catch violations.

The SPIN verification workflow consists of five main steps. First, we write the Promela model (`git_sim.pml`), defining data structures (the Commit typedef), operations (commit, merge, push), LTL properties, and assertions. Second, we generate the verifier by running `spin -a git_sim.pml` to produce `pan.c`. Next, we compile it with `gcc -O2 -o pan pan.c`. Fourth, we run `./pan -a -m100000` to explore all states, check properties, and report violations. Finally, we analyze the results: for our extended model, SPIN explored 192 million states using 30GB memory in 7 minutes, verifying all properties.

B. Properties Verified

We verify six LTL properties in total. Table VI lists all properties.

Property	Description	Model
no_cycles	Parent commits have lower IDs than children	Both
valid_heads	Branch HEADs reference commits that exist in repo	Both
merge_structure	Merge commits have exactly two parents	Both
valid_active_branch	Active branch index stays within bounds	Both
conflict_only_in_merges	Conflicts only occur in merge commits	Ext.
valid_file_modifications	File IDs within range (1 to NFILES)	Ext.

TABLE VI: Verified LTL properties

All properties are expressed as LTL formulas. SPIN converts each formula into a never claim automaton. The verifier checks that the system never violates the property. Our verification process involved complex checking each property individually, followed by complete safety verification.

VI. DISCUSSION OF THE RESULTS

A. Base Model Results

We ran verification on the base `git_sim.pml` model. Table VII shows the results.

Metric	Value
States explored	62,470,724
States matched	68,334,398
Transitions	130,805,120
Memory used	8,691 MB
Verification time	100 seconds
Depth reached	3,915
Properties verified	4/4
Errors found	0

TABLE VII: Base model verification results

The verification explored over 62 million unique states. Memory compression reduced storage to 81% of uncompressed size. All four properties passed without errors. The depth of 3,915 shows that SPIN explored long execution sequences.

Specifically, the properties `no_cycles`, `valid_heads`, `merge_structure`, and `valid_active_branch` were all satisfied, ensuring that the commit graph remains acyclic, branch heads always point to existing commits, merge commits have a proper parent structure, and there is always a unique active branch throughout the execution.

B. Extended Model Results

We utilize verification on `git_sim_conflicts.pml` with file tracking and conflict detection. Table VIII shows the results.

Metric	Value	vs Base
States explored	192,211,280	3.1×
States matched	211,596,550	3.1×
Transitions	403,807,830	3.1×
Memory used	29,882 MB	3.4×
Time	417 sec (7 min)	4.2×
Depth reached	5,516	1.4×
Properties	6/6	+2
Errors	0	–

TABLE VIII: Extended model results

The extended model explored 1.92×10^8 states in under seven minutes, with state compression achieving approximately 83% memory efficiency. Both additional correctness properties, `conflict_only_in_merges` and `valid_file_modifications`, were satisfied without counterexamples.

The state space grew by roughly a factor of three compared to the base model, which is consistent with the introduction of file-level tracking and additional non-deterministic choices. The maximum search depth increased to 5,516, reflecting the presence of more complex execution paths in the extended system.

C. Coverage Analysis

SPIN provides exhaustive state exploration within the constrained parameters. This means every possible execution sequence within the model constraints is tested. Specifically, the verifier explores:

- 1) **All interleavings of operations across two repositories**, capturing every possible scheduling of concurrent actions.
- 2) **All orderings of network message deliveries**, modeling arbitrary delays and reorderings in the communication layer.
- 3) **All non-deterministic choices**, including which operation is executed next and which file is modified.
- 4) **All reachable states up to depth 5,516**, ensuring that long execution prefixes are explored exhaustively within this bound.

Traditional testing cannot provide this coverage. A random test would need billions of runs to match this exhaustive exploration. Model checking guarantees that if a property violation exists within our bounds, SPIN will find it.

The "unreached states" in verification output show some code paths were not taken. This is expected. For example, merge operations are not always chosen. Some specific interleavings never occur due to synchronization constraints. However, all critical paths for the verified properties were explored.

VII. DISCUSSION

A. Insights and Implications

The verification confirms that the modeled DVCS operations maintain consistency under all execution scenarios. All six properties maintained across 192 million states in the extended model.

Adding content-aware conflict detection increased the state space by approximately 3×. This is reasonable increase considering we added file tracking with non-deterministic file choices. The verification remained tractable.

The fact that all properties held shows the model is internally consistent. If we had made errors in the operation logic, SPIN would have found counterexamples. For example, if push operations incorrectly updated branch pointers, the `valid_heads` property would have failed.

B. Limitations and Threats to Validity

Our model has several limitations:

Bounded parameters: We verify only 3 commits and 3 operations per repository. Real systems have thousands of commits. However, concurrency bugs typically manifest in small scenarios. Our bounds are sufficient to catch fundamental design errors.

Simplified conflict model: We detect file-level conflicts only. Real Git performs line-level three-way merges. Semantic conflicts are not modeled. Extending to line-level merging would increase state space significantly.

Abstracted operations: We omit rebase, cherry-pick, and other advanced operations. These are important for real workflows but would expand the model considerably.

No network failures: We assume messages eventually deliver. Real networks can lose packets. Modeling failures would require timeout and retransmission algorithm.

State space explosion: Even with small bounds, we reach 192 million states. Larger models are computationally infeasible without abstraction techniques.

Despite these limitations, our model captures the essential concurrent behaviors of DVCS systems. The verified properties provide meaningful guarantees.

C. Lessons Learned

Parameter tuning requires iteration: Our first attempt with MAX_COMMITS=5 failed completely. We learned that you cannot simply pick "reasonable" parameters. State space grows exponentially, a well-known challenge in model checking[11]. You must test and adjust based on actual verification results. The tuning process took several iterations but taught us how several parameters affect state space.

Topological ordering simplifies verification: Using sequential commit IDs (1, 2, 3...) guarantees acyclicity by construction. A commit cannot reference a parent with a higher ID because that parent does not exist yet. This eliminates the need for cycle detection algorithms. It is both elegant and practical.

Promela quirks require workarounds: Promela does not support true multidimensional arrays. We had to flatten 2D arrays into 1D and use macros like $\text{HAS}(r,c) = \text{repo_has}[r*\text{MAX_COMMITTS} + c]$. This was initially confusing but works well once you understand the pattern.

Monitor process catches errors early: Running invariant checks concurrently means SPIN finds violations as soon as they occur. This makes debugging much easier. Without the monitor, we would only see "property violated" without knowing when or how.

Choose what to model carefully: We excluded rebase and cherry-pick operations. Including them would have added significant complexity. The key is to model enough to test the core properties you care about but not so much that verification becomes impractical.

Non-determinism explores all possibilities: The non-deterministic choice in repository operations feels redundant at first. Why not specify exactly what happens? But this is the power of model checking. SPIN tries all possible choices and verifies that properties hold in every case.

D. Comparison with Testing

Traditional testing exercises selected scenarios of the concrete implementation, but cannot cover all interleavings or message orderings. Model checking, in contrast, exhaustively explores all behaviours of the abstract model within given bounds and provides a formal proof of correctness relative to the specified properties[12].

Testing is essential: it scales to the full implementation without artificial bounds, operates on real code, and can capture performance and system-integration issues. The two techniques are therefore complementary: model checking validates the design, while testing validates the implementation against that design. This integration of formal methods with traditional development practices has proven effective in critical systems[13].

VIII. FUTURE EXTENSIONS

Several directions could extend this work:

A. Three-Way Merge Algorithm

Current model detects conflicts but does not resolve them. Real Git uses three-way merge with a common ancestor. Modeling this would require tracking file content at the line level. This would enable verification of merge correctness properties.

B. Additional Operations

Important operations are not yet modeled:

- **Rebase:** Replay commits on a new base
- **Cherry-pick:** Apply specific commits to another branch
- **Stash:** Temporarily save uncommitted changes
- **Pull:** Fetch and merge in one operation

These operations have complex interactions with the commit graph. Verifying their correctness would strengthen confidence in DVCS implementations.

C. Conflict Resolution Strategies

Different merge strategies exist: recursive, ours, theirs, octopus. Each has different semantics. Modeling these strategies would enable comparative verification of their correctness guarantees.

D. Larger Scale

Increasing model scale is challenging but valuable:

- More repositories (3+) for true distributed scenarios
- More branches to test complex branch hierarchies
- More commits to verify deeper DAG properties
- Partial order reduction techniques to manage state space

Symmetry reduction could help. Many states are equivalent up to repository renaming. Exploiting this symmetry could reduce state space significantly.

IX. CONCLUSION

This paper presented a formal verification of DVCS operations using SPIN model checker. We built a base model that verified structural properties of commits, branches, and merges across two repositories. We then extended the model with content-aware conflict detection using file-level modification tracking.

Exhaustive state space exploration verified all correctness properties. The base model explored 62 million states. The extended model verified 192 million states with conflict detection. No errors were found in either model. All six LTL properties were held across all execution scenarios.

The verification process involved complex parameter tuning. Our initial attempt with larger bounds exceeded 500 million states and failed. Through systematic iteration, we found parameters that balance verification thoroughness with computational tractability.

This work accomplished:

- A formal Promela model of Git-like DVCS operations (commits, branches, merges, push)

- Verification of six correctness properties including structural invariants
- Content-aware extension with file tracking and conflict detection
- Exhaustive exploration of 192 million concurrent execution scenarios
- Analysis of parameter tuning trade-offs

The experience showed both the power and limitations of model checking. SPIN can exhaustively verify properties within bounded parameters. This provides stronger guarantees than testing. However, state space explosion limits model size. Careful parameter selection is essential.

Future extensions could add three-way merge algorithms, additional operations like rebase, and larger parameter bounds using reduction techniques. The model could be extended to verify semantic conflict detection beyond textual conflicts.

Model checking complements traditional testing. Testing validates implementation against specifications. Model checking verifies the specification itself. Both are valuable for building reliable systems.

REFERENCES

- [1] N. N. Zolkifli, A. Ngah, and A. Deraman, "Version control system: A review," *Procedia Computer Science*, vol. 135, pp. 408–415, 2018. The 3rd International Conference on Computer Science and Computational Intelligence (ICCSCI 2018) : Empowering Smart Technology in Digital Era for a Better Life.
- [2] G. M. Ghodke and T. Chavan, "An overview of git," *International Journal of Scientific Research in Modern Science and Technology*, vol. 3, p. 17–23, Jun. 2024.
- [3] M. Owahdi-Kareshk, S. Nadi, and J. Rubin, "Predicting merge conflicts in collaborative software development," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–11, 2019.
- [4] F. Zhu, X. Xie, D. Feng, N. Meng, and F. He, "On the methodology of three-way structured merge in version control systems: Top-down, bottom-up, or both," *Journal of Systems Architecture*, vol. 145, p. 103011, 2023.
- [5] G. Holzmann, "The model checker spin," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.
- [6] G. Vale, C. Hunsen, E. Figueiredo, and S. Apel, "Challenges of resolving merge conflicts: A mining and survey study," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4964–4985, 2022.
- [7] G. Cavalcanti, P. Borba, and P. Accioly, "Evaluating and improving semistructured merge," *Proc. ACM Program. Lang.*, vol. 1, Oct. 2017.
- [8] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford, "Verifying strong eventual consistency in distributed systems," *Proc. ACM Program. Lang.*, vol. 1, Oct. 2017.
- [9] C. Sun, D. Sun, Agustina, and W. Cai, "Real differences between OT and CRDT for co-editors," *CoRR*, vol. abs/1810.02137, 2018.
- [10] T. Blau, "Verifying strong eventual consistency in δ -crdts," *CoRR*, vol. abs/2006.09823, 2020.
- [11] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.
- [12] D. Peled, *Software Reliability Methods*. Springer, 2001.
- [13] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, Oct. 2009.