TORTE: Reproducible Feature-Model Experiments à la Carte

Elias Kuiter
kuiter@ovgu.de
University of Magdeburg
Magdeburg, Germany

Abstract

Feature modeling is crucial to understand and manage the variability in configurable software systems. However, existing tools for the automated analysis of feature-model formulas still face significant challenges, especially when applied to large and evolving codebases such as the Linux kernel's. In this paper, we contribute TORTE, an open-source tool that addresses some of these challenges. To this end, TORTE integrates existing tools and original contributions into a single experimentation platform. Thus, TORTE facilitates reproducible and flexible feature-model experiments, which we showcase in various successful research evaluations. In particular, TORTE finally enables researchers to analyze almost the entire history of the Linux kernel's feature model, among other systems.

CCS Concepts

• **Software and its engineering** → **Software product lines**; **Software evolution**; • **Theory of computation** → *Automated reasoning*.

Keywords

feature modeling, experimentation, satisfiability solving

ACM Reference Format:
Elias Kuiter. 2025. TORTE: Reproducible Feature-Model Experiments à la Carte. In *Proceedings of 48th International Conference on Software Engineering (ICSE 2026).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Modern software systems are often highly configurable, for example due to different and changing customer needs and regulations [8]. To systematically describe and document the variability of such systems, feature models are widely used [3]. In the past twenty years, much research has focused on enabling automated analyses of feature models, for example with SAT [2] and #SAT solvers [12]. Such analyses support a wide range of development tasks [13].

Despite their success, feature-model analyses remain technically challenging in practice on real-world systems [4]. To illustrate some of these challenges, we show a typical feature-model analysis pipeline in Figure 1, which consists of several steps. First, a stakeholder formulates an analysis question (e.g., "How configurable is

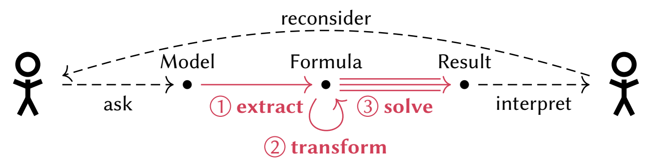


Figure 1: Simplified feature-model analysis pipeline, highlighting automated steps that we target with our tool TORTE.

this feature model?" [9]). To answer such a question with solvers, feature models must first be encoded as propositional formulas. Extracting (①) such formulas from real-world configuration languages can be challenging: For example, the KCONFIG language, which is used extensively in the Linux kernel and other system software, has complex and evolving syntax and semantics. Thus, different KCONFIG extractors make different trade-offs, which can lead to contradictory results [9]. Another crucial step is to transform (②) feature-model formulas into conjunctive normal form (CNF). Although indispensable, this step is often undocumented or oversimplified in tools and research papers, even though it can affect results [7]. Finally, there is often a wide variety of concrete solvers (③) to choose from, which affect results as well [9]. Currently, there is no interoperable and reusable way to easily experiment with the steps ①–③ (e.g., to freely mix and match different options). Moreover, several crucial tools lack automation and reproducibility. Finally, it is currently impossible to analyze historic versions of KCONFIG-based feature models. Together, these challenges severely hamper rigorous and reproducible feature-model analysis research.

In this paper, we propose the open-source tool TORTE to address these challenges. TORTE is a unified experimentation platform for feature-model experiments. It is fully automated and reproducible, allowing researchers to control the entire analysis pipeline. It also facilitates reuse and interoperability by integrating many existing tools, which can be flexibly combined. Finally, TORTE extends these tools to allow for the analysis of feature-model histories, which opens up new avenues for research. Specifically, we contribute:

- We identify open challenges in feature-model analysis tooling.
- We introduce TORTE, discuss its design and the features we offer, and we highlight how we address the identified challenges.
- We discuss our experiences applying TORTE in several publications, demonstrating its maturity and real-world applicability.

TORTE is publicly available on GitHub[1a] and archived on Zenodo (cf. Table 2).[1b] We also offer a short video[1c] that showcases TORTE.

2 Addressed Challenges

With TORTE, we address the following challenges, which arise from several limitations imposed by current feature-model analysis tools.

[1](a) github.com/ekuiter/torte, (b) 10.5281/zenodo.17940304, (c) youtu.be/VtaJiC_b8RA

C_1 History and Hierarchy Extraction It is currently not feasible to extract feature-model formulas for the entire history of long-running KConfig-based projects (e.g., the Linux kernel), which impedes research evaluations that study evolution. This limitation has two reasons: First, the syntax and semantics of the KConfig language and implementation evolve. That is, new keywords are regularly added to the KConfig language, and the existing semantics change occasionally.[2] Current extractors (e.g., KConfigReader or KClause) do not account for such changes, because they are targeted at specific versions of KConfig. Second, the KConfig language and implementation not only evolve over time—they also diverge across projects. That is, different projects use different versions of the kernel's KConfig language and implementation, and they sometimes make their own adjustments. Thus, extracting feature models for non-Linux projects is also non-trivial at times.

In addition, there is currently no functioning tool for extracting the feature hierarchy of KConfig-based feature models. Instead, existing extractors create flat models, which do not necessarily reflect developers' intentions. Instead, flatly-extracted feature models should be enriched with a hierarchy based on `menuconfig`.

C_2 Automation and Reproducibility In principle, many steps in the feature-model analysis pipeline (cf. Figure 1) are fully automatable. However, the analysis tools used in practice are sometimes outdated or assume specific execution environments. For example, the well-known KConfigReader extractor for feature-model formulas was last updated almost a decade ago. As it depends on an end-of-life version of Scala, it is technically challenging to run KConfigReader on modern systems. As another example, many (#)SAT solvers are distributed in form of precompiled binaries. These binaries are only compatible with specific operating systems (e.g., Linux) and CPU architectures (e.g., `x86` or `amd64`). Thus, it is impossible to run these solvers on many machines (e.g., macOS machines, which use `arm64`) without recompilation.

Beyond mere automation, reproducibility requires that an independent group should be able to obtain the same results using the original authors' artifacts.[3] So, artifacts must remain functional on a wide variety of machines for as long as possible. While modifying KConfigReader or recompiling solvers may momentarily improve automation, both are not sustainable solutions for reproducibility.

C_3 Reusability and Interoperability Typically, feature-model analysis artifacts are handcrafted to implement a specific experiment. This hampers later reuse or adaptation to other experimental settings, inviting the use of clone-and-own practices instead. Consequently, bug fixes and new features introduced in one artifact are not automatically propagated to others. This may lead to inconsistencies (e.g., in code or documentation) and duplicated efforts.

Furthermore, experiments can typically be conducted in various ways, which may or may not lead to equivalent results [6]. For example, different studies often apply different CNF transformations, which can significantly affect results [7]. Thus, it is desirable to be able to control such influence factors, or even transparently exchange tools (e.g., KConfigReader with KClause). However, these tools typically lack interoperability (e.g., common file formats), which impedes the comparability of results across studies.

[2]Some examples: `linux/c83f0209`, `linux/6a121588`, `linux/77a92660`
[3]https://www.acm.org/publications/policies/artifact-review-badging

Table 1: Tools integrated into and features offered by torte.

Extraction	Transformation	Solving
KConfigReader	FeatureIDE	53 SAT solvers
KClause	FeatJAR	14 #SAT solvers
ConfigFix	clausy	SATGraf
Hierarchy extractor	Z3	Jupyter notebooks
Benchmark models	CadiBack	Web dashboard
Experimentation Platform		
`Docker` containerization	Job parallelization	Stage architecture
Continuous integration	Profiling system	Experiment DSL
Reproduction packages	Resource limiting	Remote execution

Links and references to each tool are available in our GitHub repository.[1]

C_4 Ease of Use Finally, feature-model analysis artifacts typically have to orchestrate the execution of many tools (e.g., for extraction, transformation, and solving). Each of these tools may be implemented in different programming languages, assume different execution environments, and require different expertise. Consequently, creating a functional artifact takes significant knowledge and effort. As a result, knowledge is siloed within individual developers, which do not always provide thorough documentation and simple execution instructions for their artifacts. Thus, some artifacts are not well-documented or hard to use; moreover, each artifact uses a different tool orchestration approach. This can make it hard for third-party users to understand, execute, or modify artifacts.

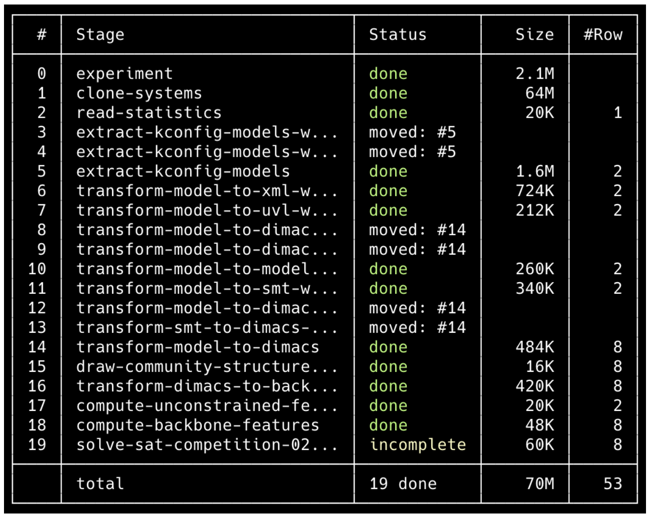
3 Introducing torte

To address the above-mentioned challenges, we contribute torte, an experimentation platform for feature-model analysis research released under the LGPL v3 license.[1] torte consists of roughly 6,000 lines of `Bash` code, which implement fully-automated *containers* for tools written in various languages (i.e., `Python`, `Java`, `Scala`, `C`, `C++`, `Rust`, and `TypeScript`). The torte platform is accompanied by several predefined *experiments*, which are executed in subsequent *stages*. Each stage performs a specific analysis step, such as *extraction*, *transformation*, or *solving*. In Table 1, we summarize the tools integrated into torte and its main features. Below, we discuss key concepts in more detail, denoting the addressed challenges.

Containers We implement torte as a set of `Bash` scripts (`.sh`), which run in- and outside of `Docker` containers. Inside of containers, we install and run various tools used to extract, transform, and solve feature-model formulas (detailed below). Outside of containers, we orchestrate the execution of containers in subsequent stages according to a user-provided experiment (also discussed below).

We choose the `Bash` script language for several reasons: First, `Bash` is widely available, which means that we can make minimal assumptions about the host environment (C_2).[4] Second, `Bash` is explicitly designed to orchestrate the execution of other tools, so it has powerful tools for inter-process communication. Third, many of the integrated tools already use `Bash` to some degree, making them easier to integrate into a `Bash`-based system. Finally, `Bash` interacts

[4]Besides `Docker`, torte only requires `Git`, `curl`, and standard GNU tools on the host.



```
 #  Stage                          Status       Size  #Row

 0  experiment                     done         2.1M
 1  clone-systems                  done          64M
 2  read-statistics                done          20K     1
 3  extract-kconfig-models-w...    moved: #5
 4  extract-kconfig-models-w...    moved: #5
 5  extract-kconfig-models         done         1.6M     2
 6  transform-model-to-xml-w...    done         724K     2
 7  transform-model-to-uvl-w...    done         212K     2
 8  transform-model-to-dimac...    moved: #14
 9  transform-model-to-dimac...    moved: #14
10  transform-model-to-model...    done         260K     2
11  transform-model-to-smt-w...    done         340K     2
12  transform-model-to-dimac...    moved: #14
13  transform-smt-to-dimacs-...    moved: #14
14  transform-model-to-dimacs      done         484K     8
15  draw-community-structure...    done          16K     8
16  transform-dimacs-to-back...    done         420K     8
17  compute-unconstrained-fe...    done          20K     2
18  compute-backbone-features      done          48K     8
19  solve-sat-competition-02...    incomplete    60K     8

    total                          19 done       70M    53
```

Figure 2: Stages of the default experiment, partially executed.

well with Docker (e.g., Dockerfiles are modified Bash scripts; also, we can reuse the same scripts in- *and* outside of containers).

Moreover, we choose Docker for containerization to ensure reproducibility ($\mathbf{C}_2$). This enables us to export reproduction packages that can be run on any machine with Docker installed, regardless of the host operating system or CPU architecture.[5] Our reproduction packages include all required Docker images, which makes them fully self-contained and, thus, ready to archive on long-term archival platforms (e.g., ZENODO). To avoid depending on Docker as the only container engine, TORTE also supports Podman.

Stages Experiment execution in TORTE is divided into a number of subsequent *stages*, which are run in isolated Docker containers. In Figure 2, we show the stages #1–19 of TORTE's default experiment. Each stage performs a specific step in the feature-model analysis pipeline (cf. Figure 1). Our example starts with a stage cloning the Git repositories of systems under analysis (#1). Subsequent stages extract feature-model formulas from their KCONFIG specifications (#3–5) before transforming (#6–14) and solving them (#15–19) in various ways. Later stages may depend on any number of previous stages; and their inputs and outputs can be flexibly connected ($\mathbf{C}_3$). Ideally, each stage should represent a pure, immutable computation step (i.e., a stage should not destructively modify its inputs). We sometimes relax this rule, for example to save disk space (e.g., we move the results of stages #3–4 into stage #5 instead of copying them, see Status in Figure 2). This architecture allows us to easily track experiment process, reuse partial results, and even continue a partially executed experiment such as the one in Figure 2 ($\mathbf{C}_4$).[6] Each stage logs its own output (including any errors), as well as relevant statistics and measurements in CSV files (e.g., the runtime and result of a (#)SAT solver, counted with #Row in Figure 2).

[5]In particular, we support Apple's macOS machines, which would otherwise be incompatible with several precompiled binaries shipped in TORTE [10]. Also, macOS has a case-insensitive file system, which requires us to rewrite part of Linux's Git history.
[6]Thus, stages are analogous to Docker layers, which are also immutable and reusable.

Listing 1: An executable excerpt of the default experiment.

```bash
#!/bin/bash
TORTE_REVISION=26c0999; TIMEOUT=10; JOBS=4
[[ $TOOL != torte ]] && builtin source /dev/stdin <<<"$(curl -fsSL \
  https://raw.githubusercontent.com/ekuiter/torte/26c0999/torte.sh)"
experiment-systems() {
  add-busybox-kconfig-history --from 1_36_0 --to 1_36_1
}
experiment-stages() {
  clone-systems; read-statistics; extract-kconfig-models
  # writes the feature model of BusyBox as UVL and DIMACS
  transform-to-uvl; transform-to-dimacs
  # Counts the number of valid configurations of BusyBox
  solve-sharp-sat --timeout "$TIMEOUT" --jobs "$JOBS"
}
```

Experiments To design a new experiment, users of TORTE can rely on a declarative Bash-based DSL, which specifies the systems under analysis and the stages to be executed ($\mathbf{C}_{3-4}$).[7] TORTE ships with more than ten predefined experiments, which can be executed out of the box or adapted to a user's needs. In Listing 1, we show an excerpt of the default experiment's definition, which consists of four parts: First, we define reusable experiment parameters (e.g., a solver timeout). The next two lines make the file self-executable, downloading TORTE automatically if needed.[8] Then, we define the systems under analysis (here, one version of BusyBox). Finally, we define the stages to be executed (as shown in Figure 2). To lower the entry barrier for users even further, we also offer one-line commands to execute predefined experiments. For example, to reproduce the default experiment, a single command suffices ($\mathbf{C}_2$):

```
curl -sL https://ekuiter.github.io/torte | sh -s default
```

We offer extensive documentation[1] [10] on setup options ($\mathbf{C}_4$), and which limitations to take into account when designing experiments.

Extraction TORTE offers *end-to-end experimentation*: That is, given a system's unpreprocessed source code, TORTE automates all remaining steps to produce analysis results. Thus, it operates directly on the ground truth (i.e., a system's source code), which enables fine-grained control over the entire analysis pipeline [6] ($\mathbf{C}_3$). This also allows us to mitigate many potential threats to validity, including the pipeline's first step, extracting feature-model formulas. We revive ($\mathbf{C}_2$) and integrate three such KCONFIG-based extractors into TORTE. These can be used interchangeably, which makes them easily comparable and interoperable ($\mathbf{C}_3$). We extend and dynamically recompile the C bindings of these extractors to improve their applicability to the full Git histories of a diverse set of systems ($\mathbf{C}_1$).[9] Thus, our extensions finally enable us to extract and publish consistent feature-model formulas for two decades of the Linux kernel's history [9, 10]. We are also currently integrating our own novel hierarchy-extraction approach ($\mathbf{C}_1$).

[7]In this DSL we developed, it is possible to pass optional and named parameters more conveniently than in plain Bash, which makes the code more readable and extensible. Also, all experiments share the underlying infrastructure, which improves reusability.
[8]In fact, it is possible to copy and paste Listing 1 verbatim into a file and execute it.
[9]Currently, TORTE supports Linux, BusyBox, Buildroot, uClibc(-ng), Toybox, axTLS, EmbToolkit, L4Re, and Freetz-NG, with more system integrations planned.

Table 2: Publications that rely on the functionality of TORTE.

Publication	Venue	Primary Functionality	Artifact
Kuiter et al. [7]	ASE '22	CNF Transformation	[12a] †
Kuiter et al. [9]	TOSEM '25	Extraction, (#)SAT Solving	[12b] † [10]
Bächle et al. [1]	SPLC '25	Extraction	[12c] † ‡
Kuiter et al. [5]	ICSE '26	SAT Solving	[12d]

† ACM Badge: *Artifacts Evaluated – Reusable v1.1* ‡ Best Artifact Award

Transformation After extraction, feature-model formulas must be transformed into conjunctive normal form (CNF) [7]. For this, we integrate four tools into TORTE (i.e., FEATUREIDE, FEATJAR, CLAUSY, and Microsoft's Z3). We also implement several file format transformations to improve these tools' interoperability ($\mathbf{C}_3$). Moreover, our efforts led to several bug reports and fixes in Z3.[10]

Solving Finally, TORTE integrates dozens of (#)SAT solvers from multiple sources (cf. Table 1), which can be compared in terms of their runtime and results ($\mathbf{C}_3$) [7]. We also integrate specialized tools for extracting backbones (i.e., CADIBACK) and visualizing formulas (i.e., SATGRAF). To summarize results, we provide a simple web dashboard that visualizes basic metrics ($\mathbf{C}_4$).[11] For advanced analyses, TORTE can also execute user-provided JUPYTER notebooks.

4 Experiences with TORTE

A prototype version of TORTE was originally developed as a single artifact to support our own research on feature-model analysis [7]. However, we quickly identified several challenges (cf. Section 2), which we wanted to address by creating a reusable experimentation platform. Since then, we have continuously improved TORTE, and applied it successfully in our own and external research (cf. Table 2), which has been certified by several ACM *Reusable* badges.[3]

Despite its success, we are also aware of a fundamental challenge, which is the long-term maintenance of TORTE. In particular, the complex setup requirements of many third-party tools (and their evolving dependencies) sometimes lead to new build issues in our `Docker` images. For existing reproduction packages, this does not matter, because they ship with all required images. However, regular maintenance is needed to ensure that TORTE remains functional for future experiments. To support this maintenance, we have set up a continuous integration pipeline to notify us whenever a build fails. We also regularly release new versions of TORTE on `GitHub`, which include prebuilt `Docker` images that we confirmed to be functional.

5 Related Work

Numerous tools have been proposed for extracting, transforming, and solving feature-model formulas (reviewed in [7, 9]). However, none of these offer end-to-end experimentation like TORTE, which integrates the full analysis pipeline in one experimentation platform ($\mathbf{C}_{2-4}$). In particular, TORTE is the first extraction tool that accounts for the implementation-defined evolution of KCONFIG ($\mathbf{C}_1$).

Schmid et al. [11] introduce *experimentation workbenches*, which TORTE is an instance of. While other workbenches [11] target static analysis or sampling, TORTE focuses on feature-model analysis, specifically, and on the precise control of experimental conditions.

6 Conclusion

Feature modeling guides developers of configurable software systems, and it enables various analyses to support related tasks. Currently, research on feature-model analysis is still challenged by limited reproducibility and interoperability, among others. To address these challenges, we proposed TORTE, an open-source experimentation platform for feature-model analysis. TORTE has already significantly contributed to several works of research. Our vision is to further consolidate TORTE as a shared platform for researchers. We also aim to provide practitioners with a pragmatic basis for applying analyses in real-world systems. For example, we aim to add support for *continuous extraction*,[13] which adds the power of feature-model analyses to existing continuous integration pipelines.

Acknowledgments

Many thanks to Eric Ketzler, Rami Alfish, and Lukas Petermann (students from the University of Magdeburg) for their diligent project work on TORTE; to Sebastian Krieter, Paul Blockhaus, Tim Bächle, and Erik Hofmayer for their feedback; and to Paul Gazzillo and Nikolaj Bjørner for their support with KCLAUSE and Z3.

References

[1] Tim Bächle, Erik Hofmayer, Christoph König, Tobias Pett, and Ina Schaefer. 2025. Investigating the Effects of T-Wise Interaction Sampling for Vulnerability Discovery in Highly-Configurable Software Systems. In *SPLC*. ACM, 45–56.
[2] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–708.
[3] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *VaMoS*. ACM, 7:1–7:8.
[4] Elias Kuiter. 2025. Towards Effective and Efficient Feature-Model Analyses for Evolving System Software. In *SPLC*. ACM, 6–13.
[5] Elias Kuiter, Urs-Benedict Braun, Thomas Thüm, Sebastian Krieter, and Gunter Saake. 2026. Can SAT Solvers Keep Up With the Linux Kernel's Feature Model?. In *ICSE*. ACM, New York, NY, USA. To appear.
[6] Elias Kuiter, Tobias Heß, Chico Sundermann, Sebastian Krieter, Thomas Thüm, and Gunter Saake. 2024. How Easy is SAT-Based Analysis of a Feature Model?. In *VaMoS*. ACM, 149–151.
[7] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *ASE*. ACM.
[8] Elias Kuiter, Jacob Krüger, and Gunter Saake. 2021. Iterative Development and Changing Requirements: Drivers of Variability in an Industrial System for Veterinary Anesthesia. In *VariVolution*. ACM, 113–122.
[9] Elias Kuiter, Chico Sundermann, Thomas Thüm, Tobias Heß, Sebastian Krieter, and Gunter Saake. 2025. How Configurable is the Linux Kernel? Analyzing Two Decades of Feature-Model History. *TOSEM* 35, 1 (Dec. 2025).
[10] Elias Kuiter, Chico Sundermann, Thomas Thüm, Tobias Heß, Sebastian Krieter, and Gunter Saake. 2025. How Configurable is the Linux Kernel? Analyzing Two Decades of Feature-Model History – RCR Report. *TOSEM* (2025). To appear.
[11] Klaus Schmid, Sascha El-Sharkawy, and Christian Kröher. 2019. *Improving Software Engineering Research Through Experimentation Workbenches.* Springer, 67–82.
[12] Chico Sundermann, Michael Nieke, Paul Maximilian Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2021. Applications of #SAT Solvers on Feature Models. In *VaMoS*. ACM, Article 12, 10 pages.
[13] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *CSUR* 47, 1 (2014), 6:1–6:45.

[10] https://github.com/Z3Prover/z3/issues/6577
[11] https://elias-kuiter.de/torte-dashboard/
[12] (a) https://doi.org/10.5281/zenodo.6525375, (b) 8190055, (c) 15647963, (d) 16084321
[13] https://github.com/acherm/24RWVMANYU-VaMoS-MODEVAR