ddueruem: A Wrapper for Feature-Model Analysis Tools

Tobias Heß
University of Ulm
Germany
tobias.hess@uni-ulm.de

Tobias Müller
University of Ulm
Germany

Chico Sundermann
University of Ulm
Germany

Thomas Thüm
University of Ulm
Germany

ABSTRACT

We present ddueruem, a tool for interfacing with BDD libraries, d-DNNF compilers, uniform, and t-wise samplers. To ease usage by researchers and practitioners, ddueruem is capable of automatically installing all tools it interfaces with, provides a coherent but customizable command-line interface to all the tools, unifies their outputs, and gathers additional statistics. In its present form, ddueruem supports the BDD libraries BuDDy and CUDD, five uniform samplers, as well as proof-of-concept support for the t-wise sampler YASA and the d-DNNF compiler d4.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**.

KEYWORDS

feature-model analysis, benchmark, knowledge compilation, sampling

ACM Reference Format:
Tobias Heß, Tobias Müller, Chico Sundermann, and Thomas Thüm. 2022. ddueruem: A Wrapper for Feature-Model Analysis Tools. In *26th ACM International Systems and Software Product Line Conference - Volume B (SPLC '22), September 12–16, 2022, Graz, Austria.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3503229.3547032

1 INTRODUCTION

Product-line engineering and therefore feature-model analysis has been applied to a multitude of domains [30] each of which yields models of different structure, size, or complexity. As this heterogeneity potentially causes approaches to scale for some but not all available models [12, 16, 30] great care needs to be taken to assure the external validity of experiments [26]. Consequently, reproduction and replication of previous results are intrinsic for evaluations in feature-model analysis and for establishing a meaningful baseline.

However, the use of third-party tools is not always straightforward. First, tools may not work as a black box but have to be used programmatically. Not only does this increase the effort required to use the tool but an improper use can also potentially introduce substantial performance penalties. A prime example is the BDD library CUDD, which, for instance, requires users to free references to BDD nodes themselves. Failing to do so potentially leads to memory blowups, greatly impacting the overall performance of the library. Second, tools are often highly configurable and may require expert knowledge to be used efficiently. Third, tools may require different formats as input and format their output differently. For example, some uniform samplers report only the selected features per sample configuration, some a full list of selected and deselected features, and some a bitmap for each configuration.

We built our tool ddueruem[1] with the goal of addressing the aforementioned issues as well as providing useful additional features. It is our vision to provide an accessible, extensible, and simply useful Python-based framework for knowledge compilation (i.e., replacing many solver calls by a one-time computation of an easy-to-query data structure) and sampling that aids researchers and practitioners in conducting experiments with high validity.

All tools and libraries ddueruem interfaces with can be used via a unified command-line interface (CLI), essentially making them black boxes. The CLI of ddueruem was designed to allow users of all maturity levels to use it. Novice users can rely on sensible default values, while expert users can tweak configurable parameters as if they were using the respective tool or library programmatically. Furthermore, ddueruem is capable of automatically installing all the tools it interfaces with, removing the need for bloated replication packages with dubious license interactions.

The remainder of this work is structured as follows. Section 2 provides foundational information on the data structures and outputs generated by the interfaced tools. We present key features and principal design choices of ddueruem in Section 3 and the integration of the four currently supported tool groups in Section 4. Finally, we discuss related tools and frameworks in Section 5, future plans for ddueruem in Section 6 and conclude in Section 7.

2 BACKGROUND

In this section, we provide foundational information on the data structures and outputs the interfaced tools generate.

2.1 Binary Decision Diagrams

Binary decision diagrams (BDD) represent Boolean functions as directed acyclic graphs [5], with two terminal nodes $\boxed{0}$ and $\boxed{1}$.

[1] https://github.com/h3ssto/ddueruem

SPLC '22, September 12–16, 2022, Graz, Austria

ACM ISBN 978-1-4503-9206-8/22/09.
https://doi.org/10.1145/3503229.3547032

Every non-terminal node is associated to a variable x of the Boolean function represented by the BDD and has precisely two outgoing edges. The low edge denotes the assignment of false to x, whereas the high edge denotes the assignment of true to x. Therefore, every path from a root node to the terminal nodes denotes a variable assignment which either satisfies the represented function (if the path ends in $\boxed{1}$) or not (if the path ends in $\boxed{0}$). Typically, one is interested in reduced and ordered BDDs and we refer to Bryant [5] for a definition of these properties.

BDDs, once computed, allow for a great number of feature-model analyses to be conducted efficiently, for example model counting [29], uniform sampling [14], and computing the semantic difference of feature-model versions [1], making them a desirable target for knowledge compilation [8]. We discuss the BDD-related features of ddueruem in Section 4.1.

2.2 Deterministic Decomposable Negation Normal Form

Boolean expressions are in negation normal form, when the only operators are *and* ($\wedge$), *or* ($\vee$), and *negation* ($\neg$) and negations only appear directly in front of literals. Deterministic, decomposable negation normal forms (d-DNNF) are a specialization of negation normal forms and we refer to Darwiche and Marquis [8] for a definition of *deterministic* and *decomposable*.

d-DNNFs excel for applications related to model counting [29], such as uniform sampling and currently scale to all but the most complex real-world feature models [30]. While ddueruem's support for d-DNNFs is currently limited, we plan to greatly expand it in the future and discuss our plans in Section 6.

2.3 Uniform Sampling

For many applications, such as testing [11, 21] and performance analysis [13, 20], one is interested in sample configurations which are representative of the solution space. In uniform sampling, every valid configuration has the same probability to be selected for the sample [2]. As a consequence, uniform samples retain the commonality of the features (i.e., if a feature is selected in m of n configuration then this ratio is maintained in the sample, provided the sample size is not too small).

ddueruem currently interfaces with five uniform samplers and verifies the uniformity of their outputs by comparing the feature commonalities in the sample with those of the model.

2.4 T-Wise Sampling

Configurable systems typically induce too many potential products to allow for exhaustive testing [16]. One is therefore interested in finding a set of configurations that is small enough to be tested exhaustively. Opposed to uniform sampling, one is not primarily interested in finding representative configurations but in covering all t-wise feature interactions. This means that all valid combinations of t features appear in at least one sample configuration.

Currently, ddueruem supports only the t-wise sampler YASA [16] due to its performance advantage over other t-wise samplers [16].

3 PRINCIPAL FEATURES

In this section, we give an overview of the key features and principal design decisions of ddueruem. Specific features for compiling BDDs and uniform sampling are discussed in Section 4.

3.1 Automated Setup

Acquiring, setting up, and configuring third party tools can be a cumbersome aspect of empirical research and practice. ddueruem addresses this nuisance by providing download and setup scripts for all tools it interfaces with, reducing their setup to a call of:

```
./setup.py <tool-name>
```

Some tools are not provided standalone but require the setup scripts to also provision additional tools or the amending of hard coded links. In some cases, ddueruem even fixes bugs,[2] we encountered when integrating the tools.

3.2 Parsing and Format Conversion

Currently, ddueruem supports the widely-used DIMACS format and the emerging UVL format [27, 28] in scenarios in which it has to parse the input itself (e.g., BDD compilation, see Section 4.1). Furthermore, ddueruem interfaces with the FeatureIDE library [15, 17] to convert different feature-model formats into another.

3.3 Benchmarking

As we mentioned before, ddueruem was build with experiments and evaluations in mind. This goal manifests itself in three key features. First, we have the vision on conducting a fully fledged evaluation with a single command, including the repetition of different aspects of the experiment (see Section 4.1 for an example). To mitigate the risk of data loss on interruption, ddueruem maintains a shadow file, allowing it to resume an interrupted experiment. Second, ddueruem tracks and measures many data points during execution and provides them in CSV files ready to be visualized or analyzed further. Users may even specify their own output templates utilizing the Jinja template language[3] to only keep track of the data they require. Third, ddueruem aims at easing the full utilization of current hardware by running experiments in parallel where possible, if the user wishes to do so.

4 SUPPORTED TOOLS

In this section, we briefly describe the tooling ddueruem interfaces with. In Section 4.1, we discuss the features of ddueruem for compiling BDDs and evaluating the performance of BDD libraries. In Section 4.2 we depict the ease of using uniform samplers with ddueruem.

4.1 BDD Compilation

ddueruem started out with the goal of providing a unified, Python-based API to the two most-widely used BDD libraries, BuDDy[4] and CUDD[5]. Rather than providing bindings to all of the respective functionalities of both libraries, which would have meant to have

[2] e.g., https://github.com/meelgroup/KUS/issues/2
[3] https://jinja.palletsprojects.com/en/3.1.x/
[4] http://buddy.sourceforge.net
[5] https://davidkebo.com/source/cudd_versions/cudd-3.0.0.tar.gz

Compute 100 random initial variable orderings per model

```
./ddueruem.py *.dimacs --bdd dvo:sift,sift_conv svo:random svo_n:100 svo_keep t:5m threads:16
```

Figure 1: CLI Command for Example 4.1

divergent APIs for each library, we opted for a unified API, encompassing a cutting set of functionalities. Our API allows to construct, manipulate, dynamically reorder, and export BDDs to a common format, regardless of the library used. Additional functionalities which are not part of the unified API may be accessed using the provided utility functions. We use the `ctypes`[6] library to interface directly with the shared libraries of BuDDy and CUDD.

In order to make BuDDy and CUDD usable as black boxes, ddueruem provides a CLI. For example, in order to compile a BDD for the well-known feature model *BusyBox* it suffices to execute

```
./ddueruem.py busybox.dimacs --bdd
```

Under the hood, this parses the input file, computes a random initial variable order, bootstraps the default BDD library (CUDD), and sets the dynamic variable ordering algorithm to the converging variant of the sift heuristic [24]. However, it is possible to finely tune every aspect of the compilation process. You want to use a pre-computed variable order? Supply it with

```
--bdd order:<your-file,...>
```

You want to timeout the BDD compilation after `t` seconds? Set a time limit with `--bdd t:<t>`. We provide an extensive documentation, outlining all the tunable parameters and their potential consequences.

As ddueruem was build with benchmarking in mind, you can conduct a fully-fledged evaluation of BDD performance with a single CLI command. Consider the following example.

Example 4.1. Suppose you want to compare the performance of the one-shot sifting dynamic variable reordering heuristic (`sift`) [23] to the converging one (`sift-conv`) [23] for every feature model in your arsenal. In order to reduce the impact of the initial variable ordering, you chose to repeat the experiment 100 times per reordering heuristic, using the same 100 random initial variable ordering for each heuristic. Finally, you want to timeout each BDD compilation after 5 minutes and use 16 threads for the evaluation, to keep the required time reasonable. Figure 1 depicts the CLI command to facilitate this evaluation.

During its execution, ddueruem will log many data points that could, potentially, be of interest for the discussion of the conducted experiment. This includes, but is not limited to, the required time per BDD-compilation attempt (or the percentage of processed clauses in case of a timeout), the time spent for dynamic reordering, and the size of the BDD throughout the compilation. The data is provided as CSV files, one per run, containing a log of the compilation process, and one containing the outcomes of all runs for an input file. As mentioned in Section 3, users may provide Jinja templates to generate custom output files.

[6] https://docs.python.org/3/library/ctypes.html

4.2 Uniform Sampling

Currently, ddueruem brings support for five uniform samplers, namely KUS [25], Quicksampler [9], SMARCH [19], SPUR [3], and Unigen [6, 7]. As all of these samplers are already usable as black boxes, ddueruem's contribution is a unified CLI for all the samplers, post-processing, verification, and a straightforward way to execute the samplers in parallel. In order to compute 100 sample configurations for the aforementioned *BusyBox* feature model, using each of the uniform samplers, it suffices to run:

```
./ddueruem.py busybox.dimacs --usample all n:100
```

Subsequently, ddueruem normalizes the outputs of the samplers into a common output format and verifies the uniformity of the samples by comparing the feature commonalities in the sample to the feature commonalities computed for the feature model. For computing the latter, ddueruem relies on the d-DNNF compiler d4, which is also used as part of the KUS uniform sampler [25].

5 RELATED WORK

In this section, we discuss tools that provide similar functionality to ddueruem and argue the need for yet another tool.

JavaBDD[7] is a well-known and widely-used (e.g., [18, 22]) framework providing Java bindings for a number of BDD libraries, namely, BuDDy, CAL, CUDD, and JDD. Like ddueruem, it provides an unified API for constructing and querying BDDs. However, JavaBDD comes without black-box capabilities, meaning that researchers and practitioners have to parse input files themselves and care for every aspect of the compilation process (e.g., static variable ordering) including the actual construction of the BDD. Furthermore, when we used JavaBDD ourselves, we encountered frequent, inexplicable crashes and a significant performance penalty when compared to ddueruem [12].

dd[8] is a framework providing Python bindings to the BDD libraries BuDDy, CUDD, and Sylvan. Opposed to ddueruem, which uses the `ctypes` library to interface with the BDD libraries, dd uses `Cython`[9]. Like JavaBDD, dd provides no black-box functionalities and, despite our best efforts and communication with the developers, we were not able to get it completely working.

BURST [2] is a framework for benchmarking uniform samplers. It offers support for ten sampling tools [2], including the five samplers ddueruem currently supports. Like ddueruem it aggregates statistics of the samplers' performance, like the time required to compute the samples. In addition, it supports the uniformity checker `barbarik`[10] to analyze the uniformity of the samples generated by the samplers. Compared to BURST, ddueruem provides a conciser CLI, normalizes the samplers' outputs, and offers to execute the

[7] http://javabdd.sourceforge.net/
[8] https://github.com/tulip-control/dd
[9] https://cython.org/
[10] https://github.com/meelgroup/barbarik

samplers in parallel, e.g., to compute samples for a number of input files.

FaMaPy[11] [10] is a framework for conducting SAT-based analyses such as core and dead feature detection on feature models. To some regard, it can be considered a reincarnation of the legacy, Java-based FAMA framework [4]. Opposed to FAMA, FaMaPy currently does not support analyses based on knowledge compilation. Therefore, the feature sets of FaMaPy and ddueruem are currently non-intersecting.

6 FUTURE WORK

For the future, we plan to enhance ddueruem along four lines. First, we plan to interface with further tools and libraries. Among others, we plan to support the d-DNNF compiler dsharp,[12] the d-DNNF framework ddnnife,[13] and our own BDD library (which currently is in active development).

Second, we aim to make ddueruem more accessible and easier to integrate into other tools. To this end, we are currently developing a RESTful API for ddueruem in order to deploy ddueruem as a microservice. Not only will this make the deployment of ddueruem to compute servers straightforward, it will also allow for a painless integration into web-based services.

Third, we plan to further enrich the reporting of measurements of ddueruem. While the ability to customize outputs using Jinja already is a step in the right direction, we want to increase ddueruem's usefulness by also providing functionality for, among others, computing statistical significance and visualizing data.

Fourth, we envision to fully close the loop to feature-model analysis. While currently the focus of ddueruem lies on knowledge compilation and sample generation, we aim to provide functionality to also *use* the artifacts and samples to answer feature-model analysis queries.

7 CONCLUSION

We presented our tool ddueruem which enables researchers and practitioners to access knowledge compilers and uniform samplers through a concise and streamlined command-line interface. In addition, our tool is capable of installing all the tools it interfaces with and even applies patches where necessary. Whenever possible, our tool aims to unify the output of the interfaced tools to the same format, in order to ease successive data analysis. Last, our tool comes with benchmarking in mind and has the ability to perform a fully-fledged evaluation with a single command-line query.

ACKNOWLEDGMENTS

We like to thank Sabrina Böhm for her input and the participants of the FOSD community meeting for the lively discussions and feature suggestions.

REFERENCES

[1] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. 2012. Feature Model Differences. In *CAiSE*. Springer, 629–645.

[2] Mathieu Acher, Gilles Perrouin, and Maxime Cordy. 2021. BURST: A Benchmarking Platform for Uniform Random Sampling Techniques. In *SPLC*. ACM, New York, NY, USA.

[3] Dimitris Achlioptas, Zayd S. Hammoudeh, and Panos Theodoropoulos. 2018. Fast Sampling of Perfectly Uniform Satisfying Assignments. In *SAT (LNCS)*. Springer, Berlin, Heidelberg, 135–147.

[4] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2007. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *VaMoS*. Technical Report 2007-01, Lero, 129–134.

[5] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *TC* C-35, 8 (1986), 677–691.

[6] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. 2015. On Parallel Scalable Uniform SAT Witness Generation. In *TACAS*. Springer, 304–319.

[7] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2014. Balancing Scalability and Uniformity in SAT Witness Generator. In *DAC*. ACM, New York, NY, USA, 60:1–60:6.

[8] Adnan Darwiche and Pierre Marquis. 2002. A Knowledge Compilation Map. *JAIR* 17, 1 (2002), 229–264.

[9] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient Sampling of SAT Solutions for Testing. In *ICSE*. ACM, New York, NY, USA, 549–559.

[10] José A. Galindo and David Benavides. 2020. A Python Framework for the Automated Analysis of Feature Models: A First Step to Integrate Community Efforts. In *SPLC*. ACM, New York, NY, USA, 52–55.

[11] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2019. Test Them All, Is It Worth It? Assessing Configuration Sampling on the JHipster Web Development Stack. *EMSE* 24, 2 (2019), 674–717.

[12] Tobias Heß, Chico Sundermann, and Thomas Thüm. 2021. On the Scalability of Building Binary Decision Diagrams for Current Feature Models. In *SPLC*. ACM, 131–135.

[13] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-Based Sampling of Software Configuration Spaces. In *ICSE*. IEEE, 1084–1094.

[14] Donald Ervin Knuth. 2009. *The Art of Computer Programming: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Vol. 4. Addison-Wesley.

[15] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. 2017. FeatureIDE: Empowering Third-Party Developers. In *SPLC*. ACM, 42–45.

[16] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: Yet Another Sampling Algorithm. In *VaMoS*. ACM, Article 4, 10 pages.

[17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.

[18] Marcílio Mendonça. 2009. *Efficient Reasoning Techniques for Large Scale Feature Models*. Ph. D. Dissertation. University of Waterloo.

[19] Jeho Oh, Paul Gazzillo, and Don Batory. 2019. t-wise Coverage by Uniform Sampling. In *SPLC*. ACM, 84–87.

[20] Juliana Alves Pereira, Mathieu Acher, Hugo Martib, and Jean-Marc Jézéquel. 2020. Sampling Effect on Performance Prediction of Configurable Systems: A Case Study. In *ICPE*. ACM, 277–288.

[21] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *ICST*. IEEE, 240–251.

[22] Richard Pohl, Kim Lauenroth, and Klaus Pohl. 2011. A Performance Comparison of Contemporary Algorithmic Approaches for Automated Analysis Operations on Feature Models. In *ASE*. IEEE, 313–322.

[23] Raise Language Group. 1993. *Raise Method Manual*. Prentice-Hall Inc.

[24] Richard Rudell. 1993. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *ICCAD*. IEEE, 42–47.

[25] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S Meel. 2018. Knowledge Compilation Meets Uniform Sampling. In *Proc. Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*. EasyChair, 620–636.

[26] Norbert Siegmund, Stefan Sobernig, and Sven Apel. 2017. Attributed Variability Models: Outside the Comfort Zone. In *ESEC/FSE*. ACM, 268–278.

[27] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. 2021. Yet Another Textual Variability Language? A Community Effort Towards a Unified Language. In *SPLC*. ACM, 136–147.

[28] Chico Sundermann, Tobias Heß, Dominik Engelhardt, Rahel Arens, Johannes Herschel, Kevin Jedelhauser, Benedikt Jutz, Sebastian Krieter, and Ina Schaefer. 2021. Integration of UVL in FeatureIDE. In *MODEVAR*. ACM, 73–79.

[29] 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.

[30] Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2020. Evaluating #SAT Solvers on Industrial Feature Models. In *VaMoS*. ACM, Article 3, 9 pages.

[11] https://github.com/diverso-lab/core
[12] https://github.com/QuMuLab/dsharp
[13] https://github.com/ddnnife