



UVLS: A Language Server Protocol For UVL

Jacob Loth
University of Ulm
Germany

Chico Sundermann
University of Ulm
Germany

Tobias Schrull
University of Ulm
Germany

Thilo Brugger
University of Ulm
Germany

Felix Rieg
University of Ulm
Germany

Thomas Thüm
University of Ulm
Germany

ABSTRACT

The Universal Variability Language (UVL) is a community-driven textual format for feature models. Over the last few years, UVL has been integrated into several relevant product-line tools. One of UVLs major advantages is its manual readability and editability. Still, without automated support it is hard to overview larger UVL models. We implemented a language server protocol (LSP) that includes syntactical and semantic analyses of UVL files. The LSP comes with several handy features to make textual editing of UVL more convenient, such as syntax checks and anomaly detection. Furthermore, the LSP supports the configuration of UVL files in a web-based editor with decision propagation. The reasoning engine supports constraints with a high level of expressiveness, such as numerical and string features. Due to the generic interface of LSPs, UVL can be integrated into modern IDEs with only small efforts. We already integrated the LSP into Visual Studio Code and NeoVim.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines; System modeling languages.**

KEYWORDS

feature modeling, variability language, language server protocol

ACM Reference Format:

Jacob Loth, Chico Sundermann, Tobias Schrull, Thilo Brugger, Felix Rieg, and Thomas Thüm. 2023. UVLS: A Language Server Protocol For UVL. In *27th ACM International Systems and Software Product Line Conference - Volume B (SPLC '23)*, August 28-September 1, 2023, Tokyo, Japan. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3579028.3609014>

1 INTRODUCTION

Feature models are commonly used to specify the valid configurations of product lines [12]. While feature diagrams seem to be prevalent as graphical notation, various textual formats are used in practice [3]. The multitude of formats causes overhead for researchers and practitioners.

The Universal Variability Language (UVL) is a textual specification for feature models that is developed in tight cooperation

with the community [15]. The goal of the cooperative development approach is to achieve wide adoption for UVL to simplify exchange between different tools. Recently, several popular feature-modeling tools, such as FeatureIDE [10], FLAMA [8], and TraVarT [7] integrated UVL. While the recent adoption indicates the interest in UVL, tool support specific for UVL is still limited.

One goal of UVLs language design is that UVL models should be readable and editable by humans [15]. When following these requirements, we expect the language to be more accessible and suitable for teaching. However, when dealing with larger feature models, manual editing is still hard without any editing support.

In this work, we present our language server protocol (LSP) UVLS that supports textual editing of UVL models. An LSP is an interface for language features (e.g., syntax highlighting) that is widely supported in modern IDEs and, thus can be easily integrated. We already integrated the LSP in Visual Studio Code¹ and NeoVim. For the Visual Studio Code integration, we provide a short video showcasing the features of UVLS.²

Our LSP UVLS³ comes with several features to support the editing process. First, the UVL file is checked whether it is syntactically correct according to the language specification. Second, we employ semantic analyses for detecting anomalies, such as void feature models. Third, UVLS provides a configuration editor that checks the validity of the user's feature (de-)selections.

The SMT-based reasoning engine enables the automated analysis of complex language features, such as numerical expressions and string features. Other available tools [8, 10] that perform automated analyses on UVL models only support a subset of its expressiveness as they are limited to propositional logic.

2 UNIVERSAL VARIABILITY LANGUAGE

The Universal Variability Language (UVL) is a textual representation for specifying feature models [15]. Within the MODEVAR initiative [4], UVL has been developed as a community effort towards a unified variability language [15]. The goal is to achieve wide adoption for UVL, to simplify exchange of feature models between researchers and practitioners.

UVL aims to be readable and editable by humans in the textual format. Listing 1 shows a simplified bike feature model specified in the UVL format. UVL models typically consist of at least two parts: the feature hierarchy and additional cross-tree constraints.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SPLC '23, August 28-September 1, 2023, Tokyo, Japan

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0092-7/23/08.

<https://doi.org/10.1145/3579028.3609014>

¹<https://marketplace.visualstudio.com/items?itemName=caradhras.uvls-code>

²<https://youtu.be/KhPfbPWliA>

³<https://github.com/Universal-Variability-Language/uvl-lsp>

Listing 1: Feature Model in UVL

```

features
  Bike
    mandatory
      Brake {Weight 3}
      String Manufacturer
      Integer Inch
    optional
      Bell {Weight 1}
      "Training Wheels" {Manufacturer 'Wheelio'}
      "Light System"
        or
          Front {Weight 2}
          Back {Weight 1.5}
      Dynamo {Weight 3}

constraints
  Front | Back => Dynamo
  sum(Weight) < 10
  Inch > 22 => !"Training Wheels"
  "Training Wheels" => (Manufacturer == "Training Wheels".
    ↪ Manufacturer)

```

In the following, we explain basic language design of UVL including recent advances with more sophisticated language features.⁴ The feature hierarchy is realized with indentation to mimic a feature diagram and improve readability. Each feature has one of the following domains: *Boolean*, *real*, *integer*, or *string*. In our running example, *Manufacturer* is a string feature, *Inch* is an integer feature, and all other features are Boolean. Furthermore, UVL supports the following parent-child relationships: *optional*, *mandatory*, *alternative*, *or*, and *group cardinalities* (i.e., select $[n..m]$ features). Features may also have a cardinality $[n..m]$ which means that between n and m instances of this feature can be included. Feature attributes, such as the weight of a feature, can also be attached as key-value pairs.

Cross-tree constraints with different levels of expressiveness can be specified below the tree hierarchy. First, arbitrary propositional constraints over literals corresponding to features or boolean feature attributes can be specified. In our running example, having a *Front* or *Back* light requires a *Dynamo*. Second, UVL supports numerical expressions over feature attributes and numerical features. As an example, the overall weight of bike components attached to the frame can be at most 10. Third, some basic string operations on string features and attributes are included. The *Training Wheels* of our bike example needs to be from the same manufacturer as the overall bike.

Imports. UVL supports decomposing feature models into smaller subsets. The decomposed models can then be referenced in a composed model with an import mechanism. Listing 2 shows a composed model equivalent (but shortened here) to our running example. Here, the light system is encapsulated as alone-standing file. In the main file for our bike, the *lights.uvl* submodel is imported with the alias *l* which can be used for references. The submodel can be attached anywhere in the subtree by referencing its root feature at

⁴<https://github.com/Universal-Variability-Language/uvl-parser-java>

Listing 2: Imports in UVL

```

features
  "Light System"
    or
      Front {Weight 2}
      Back {Weight 1.5}

```

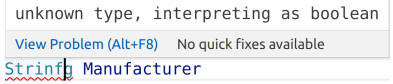


Figure 1: Syntactical Analyses with UVLS

the corresponding place. The features of the submodel can also be used in constraints by using their alias as namespace.

3 LANGUAGE SERVER PROTOCOL

Our language server protocol UVLS supports syntactical and semantic analyses for UVL models. Furthermore, UVLS comes with several convenience features to ease the textual editing. The implementation bases on tree-sitter⁵ for parsing and syntactical analysis, and on the SMT solver Z3 [6] for semantic analyses.

3.1 Syntactical Analysis

The syntactical analyses check whether the given feature model adheres to the language specification of UVL. In particular, UVLS checks whether the tree-sitter grammar⁶ accepts the input feature model. Figure 1 shows an example error message for a typo in the feature type of *Manufacturer*.

3.2 Semantic Analysis

UVLS also supports semantic analysis in terms of anomaly detection. The following analyses are currently supported: void model (i.e., zero valid configurations), dead feature (i.e., a feature appears in no valid configuration), contradiction (i.e., a constraint can never be satisfied), and tautology (i.e., a constraint is always satisfied, and thus, redundant). For the identified anomalies, explanations are shown that indicate which constraints and features are probably responsible for the anomaly. Figure 2 shows an adaptation of the bike model that is void due to a new constraint *!Brake*. The void status is annotated at the feature keyword. In addition, the feature *Brake* and the new constraint are marked to give an explanation.

The semantic analyses are realized with the SMT solver Z3 [6] and some additional preprocessings to handle string constraints. In particular, UVL models are converted to the SMTLib [2] standard and analyses are reduced to satisfiability queries. For instance, a feature f is dead if the formula conjuncted with a positive literal l_f corresponding to that feature is unsatisfiable.

3.3 Configuration Process

UVLS also provides a configuration editor to derive valid configurations for a given UVL model. The configuration editor supports

⁵<https://tree-sitter.github.io>

⁶<https://github.com/Universal-Variability-Language/uvl-lsp/blob/master/tree-sitter-uvl/src/grammar.json>

```

1  feature
2  Bike
3
4      mandatory
5          Brake {Weight 3}
6          String Manufacturer
7          Integer Inch
8
9      optional
10         Bell {Weight 1}
11         "Training Wheels" {Manufacturer 'Wheelio'}
12         "Light System"
13
14         or
15             Front {Weight 2}
16             Back {Weight 1.5}
17             Dynamo {Weight 3}
18
19 constraints
20     Front | Back => Dynamo
21     sum(Weight) < 10
22     Inch > 22 => !"Training Wheels"
23     "Training Wheels" => (Manufacturer == "Training Wheels".Manufacturer)
24     !Bike

```

Figure 2: Semantic Analyses with UVLS

OutputFile: bike.uvl.json	SAT State: UNSAT	Solver State: idle
Files:		
Path	Configuration	
> bike.uvl		
> Bike(x)	?	
< Brake(x)	?	
Manufacturer(x)	NoHelpHere	UNSAT CONFIG
Inch(x)	?	
Bell(x)	?	
"Training Wheels"(x)	true	UNSAT CONFIG
"Light System"(x)	?	

Figure 3: Configuration Process with UVLS

configuration decisions for all feature domains in UVL, namely Boolean, integer, real, and string. Feature attributes can also be configured. At each point of the configuration process, UVLS indicates whether the configuration is still satisfiable. Also, one example solution for the current partial configuration is displayed. If the decisions of the user violate the constraints of the feature model, the configuration editor also shows an explanation. Figure 3 shows the configuration editor of UVLS. Here, the user gave an invalid Manufacturer name as input which violates the cross-tree constraint "Training Wheels" => (Manufacturer == "Training Wheels".Manufacturer). The conflicting decisions are marked in the configuration editor. The configurations can be persistently saved as json-files.

The current configuration can also be shown in the feature-model editor to simplify understanding the impact of decisions. Figure 4 shows an example configuration annotated to the feature model. Each configuration decision is visualized using code inlays.

3.4 Convenience Features

UVLS comes with several handy convenience features to simplify the editing process.

Syntax Highlighting. Based on the identified tokens from the syntactical analyses, UVL applies syntax highlighting. Hereby, the

```

features
Bike : true
  mandatory
    Brake : true {Weight : 3}
    String Manufacturer : Wheelio
    Integer Inch : 1
  optional
    Bell : true {Weight : 1}
    "Training Wheels" : true {Manufacturer : Wheelio 'Wheelio'}
    "Light System" : false
    or
      Front : false {Weight : 0}
      Back : false {Weight : 0}
      Dynamo : true {Weight : 3}
  constraints
    Front : false | Back : false => Dynamo : true
    sum(Weight) < 10
    Inch > 22 => !"Training Wheels" : true
    "Training Wheels" : true => (Manufacturer : Wheelio == "Training Wheels".Manufacturer : Wheelio)

```

Figure 4: Inline Configuration with UVLS

```

Bike
  alternative
  mandatory
  optional
  or
  (Manufacturer == M)
    "Training Wheels".Manufacturer
    Manufacturer
  imports
  sub1
  sub2
  feat

```

Figure 5: Autocompletion with UVLS

color identifies grouping keywords (e.g., features or optional), feature identifiers, feature types (e.g., String), attribute identifiers, numeric constants, and string constants.

Auto Completion. UVLS supports context-sensitive auto completion. Based on the type of construct the grammar expects, the LSP suggests valid inputs. Figure 5 shows some examples on possible auto completions. First, the LSP detects that a group type is required and suggests possible options. Second, the *sum* aggregate shows all available numerical attributes. Third, all other UVL files in the project (here: sub1 and sub2) are suggested as imports.

Handling of References. Features and feature attributes are typically referenced multiple times in constraints. UVLS includes support to find other references and jump to them. In particular, this can be used to find the definition in the tree hierarchy for a feature appearing in a cross-tree constraint. Still, changing each occurrence by hand would be tedious. To simplify renamings, UVLS enables renaming all references simultaneously.

Visualization. The visualization of feature models greatly facilitates comprehension. UVLS provides a built-in feature to display Feature Models using Graphviz⁷ that bases on the visualization proposed by Kuitert et al. [9]. This will create a .dot file representing the feature model. In VSCode, the model can be directly displayed with the recommended GraphViz extension⁸. Then, you can also edit the generated file and see the changes in realtime. Figure 6 shows the visualization of our running example.

⁷<https://graphviz.org>

⁸<https://marketplace.visualstudio.com/items?itemName=tintinweb.graphviz-interactive-preview>

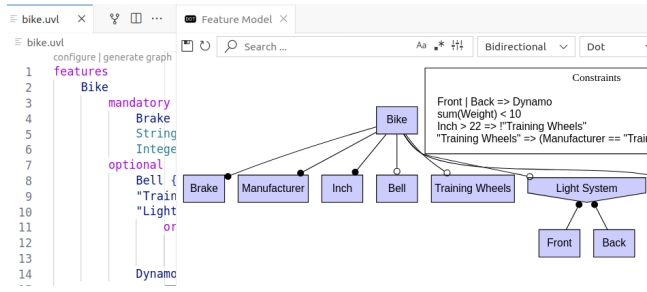


Figure 6: UVL Model Visualization

4 RELATED WORK

Tool Support for Editing UVL. In FeatureIDE [10], the graphical editor can be used to develop UVL models. FLAMA [8] can be used to load UVL models and edit them programmatically. Our LSP UVLS seems to be the only tool targeting the textual editing of UVL models and feature models in general.

UVL Parsers. Several parsers have been considered for UVL [8, 15]. Originally, the UVL was published with a Clojure-based parser library [15]. FLAMA uses an ANTLR [13]-based parser for analysis in Python [8]. Both parsers support only the *Boolean* level of UVL as the remaining extensions were developed rather recently. A recently developed parser⁹ supports the language levels but targets Java while our internal parser targets Rust.

Analysis of UVL Models. FeatureIDE [10] and FLAMA [8] both support automated analyses of UVL models. However, for both the analyses are limited to the *Boolean* level of UVL. The more complex language levels, namely *Arithmetic* and *Type*, are not supported.

Tool Support for Feature Modeling. Several tools targeting feature modeling are available [1, 5, 8, 10, 11]. Neither of these tools targets textual editing of models. Only FeatureIDE [10] and FLAMA [8] support UVL. SPLLOT [11], FLAMA [8], FeatureIDE [10], FAMILIAR [1] use only SAT-based reasoning engines and, thus, could not perform reasoning about the *Arithmetic* and *Type* level of UVL. Note that a prototypical extension to FeatureIDE provided SMT support which could be adapted to analyze the *Arithmetic* level [14]. The extension was, however, never added to a release version of FeatureIDE.¹⁰ FAMA [5] uses, amongst other reasoning engines, a CSP solver as reasoning engine which could potentially be adapted to reason on constraints from the *Arithmetic* level of UVL. However, the feature model format of FAMA does not support such constraints.

5 CONCLUSION

The Universal Variability Language (UVL) is getting more widely adopted. While one of the main goals of UVL is being readable and editable in textual form, previous tools have no support for textual editing. Our language server protocol UVLS provides several features to simplify textual editing of UVL. Furthermore, UVLS comes with a powerful reasoning engine that enables analyzing complex

constraints. Due to the widely adopted interface of language server protocols, UVL can be easily integrated into other modern IDEs.

In the future, we envision UVLS as suitable choice for learning and teaching the basics of UVL and software-product-lines development. Hence, we plan to add more educational examples and tutorials. A graphical visualization for UVL models to further improve understanding is planned. We also aim to include a simple mechanism to derive variants from an annotated source code for teaching purposes.

ACKNOWLEDGEMENTS

We thank Pascal Förster and Martin Maurer for their valuable contributions to UVLS.

REFERENCES

- [1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. Familiar: A Domain-Specific Language for Large Scale Management of Feature Models. *SCP* 78, 6 (2013), 657–681.
- [2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. Department of Computer Science, The University of Iowa.
- [3] Maurice H. ter Beek, Klaus Schmid, and Holger Eichelberger. 2019. Textual Variability Modeling Languages: An Overview and Considerations. In *SPLC*. ACM, 151–157.
- [4] David Benavides, Rick Rabiser, Don Batory, and Mathieu Acher. 2019. First International Workshop on Languages for Modelling Variability (MODEVAR 2019). In *SPLC*. 323–323.
- [5] 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.
- [6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT solver. In *TACAS*. Springer, 337–340.
- [7] Kevin Feichtinger, Johann Stöbich, Dario Romano, and Rick Rabiser. 2021. TRAVART: An Approach for Transforming Variability Models. In *VaMoS*. ACM, Article 8, 10 pages.
- [8] 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 *MODEVAR*. ACM, 52–55.
- [9] Elias Kuitert, Sebastian Krieter, Jacob Krüger, Gunter Saake, and Thomas Leich. 2021. variED: An Editor for Collaborative, Real-Time Feature Modeling. *EMSE* 26, 2 (2021), 24.
- [10] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.
- [11] Marcilio Mendonça, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *OOPSLA*. ACM, 761–762.
- [12] Marcilio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-Based Analysis of Feature Models is Easy. In *SPLC*. Software Engineering Institute, 231–240.
- [13] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A Predicated-LL(k) Parser Generator. *SPE* 25, 7 (1995), 789–810.
- [14] Joshua Sprey, Chico Sundermann, Sebastian Krieter, Michael Nieke, Jacopo Mauro, Thomas Thüm, and Ina Schaefer. 2020. SMT-Based Variability Analyses in FeatureIDE. In *VaMoS*. ACM, Article 6, 9 pages.
- [15] 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.

⁹<https://github.com/Universal-Variability-Language/uvl-parser-java>

¹⁰<https://github.com/FeatureIDE/FeatureIDE/releases>