mech-uvl: A Mechanized Semantic Toolkit for the Universal Variability Language

Frédéric Loulergue[0000-0001-9301-7829], Jolan Philippe[0000-0001-8759-4566], and Nicolas Paul[0009-0008-8835-8686]

Université d'Orléans, INSA Centre Val de Loire, LIFO UR 4022, Orléans, France

Abstract We present `mech-uvl`, a mechanized semantic toolkit for the Universal Variability Language (UVL). UVL is intended to support interoperability between feature modelling tools, but existing implementations make different semantic choices, for example in the treatment of attributes, typed features, and constraints. `mech-uvl` makes such choices explicit and executable. The toolkit combines a C# front-end, based on the official ANTLR grammar, with a formal core written in Dafny. The front-end provides preprocessing, parsing, and pretty printing for concrete UVL files, and calls executable checkers compiled from the Dafny development. This development defines a shared abstract syntax, well-formedness and typing predicates, executable checkers whose absence of error entails these predicates, type inference for introduced attributes, and language level specification inference. We describe the design methodology used to derive these definitions from the UVL grammar, the UVL literature, and selected examples from existing tools and model repositories. We also present a variability model of UVL semantic choices and show how `mech-uvl` can be used both as a practical static analysis tool and as a basis for a mechanized extensional semantics of UVL.

Keywords: Universal Variability Language · formal semantics · mechanization · static checking · software product lines

1 Introduction

Feature models are a central artefact for modelling the configuration space of software product lines [7]. Such models describe features that may or must be selected in valid products, alongside their attributes with structural relations and cross-tree constraints. Over the years, many feature-modelling languages and tools have been proposed, with different concrete syntaxes, expressiveness, and analysis backends. The Universal Variability Language (UVL) has recently emerged as a community-driven effort to support interoperability through a common textual language for feature modelling [4].

In practice, there exist many UVL implementations, but they do not follow the exact same semantics, which implies that models may be analysed under different assumptions. For example, UVLS [21] requires all attributes to be explicitly declared before they are used in constraints, whereas FlamaPy [5] allows

attributes to be introduced implicitly. Thus, the same UVL model may be accepted by one tool and rejected by another, or may be analysed under different semantic assumptions. Such flexibility is useful for a language intended to serve different tools and communities, but we argue that it should be made explicit. Before fixing a complete semantics, it is important to identify the semantic choices made by existing tools, to make these choices explicit, and to evaluate their impact on existing UVL models. This calls for executable support: candidate definitions should be applicable to concrete models, so that future semantic choices can be evaluated experimentally on UVL model repositories.

Interactive theorem provers have been successfully used to mechanize the semantics and correctness of software artefacts. However, our goal is not only to state and prove semantic properties, but also to connect them to executable tooling for concrete UVL files. Dafny [20] provides a compromise between specification, automated proof support, and compilation into executable languages. Since Dafny supports compilation to C#, it can be integrated with a UVL front-end based on the official ANTLR grammar. We therefore chose Dafny and C# as an engineering decision to minimize integration work for our prototype.

This paper presents `mech-uvl`, a mechanized semantic toolkit for UVL. The paper makes three contributions. First, we define in Dafny a shared abstract syntax for UVL together with well-formedness conditions, well-typedness conditions, type inference for introduced attributes, and language level specification inference. These definitions are written so that their executable parts can be compiled and used on concrete models. The executable checkers are related to the logical predicates by Dafny postconditions: when the core, typing, or level checker reports no error, the corresponding predicate holds for the model environment. Second, we connect this formal core to a C# front-end based on the official UVL ANTLR grammar. The front-end preprocesses, parses, and pretty-prints UVL files, and calls the executable Dafny checkers. Third, we identify several semantic variation points in UVL, in particular around typed features, undefined constraints, and attributes introduced through constraints, and make these choices explicit as a variability model. This makes `mech-uvl` both a practical static-analysis tool for existing UVL models and a basis for the ongoing development and experimental comparison of candidate UVL semantics.

The remainder of the paper is organized as follows. Section 2 gives an overview of `mech-uvl` and its architecture. Section 3 presents the methodology and design choices that guided the formalization. Section 4 describes well-formedness, type checking, type inference, and language level checking. Section 5 discusses related work. Section 6 discusses current limitations and the ongoing work towards an extensional semantics. Finally, Section 7 concludes and outlines future work. The version of the toolkit presented in this paper is available [22].

2 Overview of `mech-uvl` and its Architecture

`mech-uvl` is a tool-supported formalization of the Universal Variability Language. It takes a UVL model as input and provides a toolkit, as a collection of semantic definitions and executable checkers whose, for processing and reasoning about UVL models. An overview of the overall architecture of the toolkit's components is given in Figure 1 and an example of a UVL model is given in Figure 2. `mech-uvl` provides a mechanized semantics of UVL defined in a theorem prover. The toolkit relies on two languages. File handling, preprocessing, parsing, and pretty-printing are implemented in C# (2905 LoC) while the language definition and semantic checks are implemented in Dafny (5789 LoC).

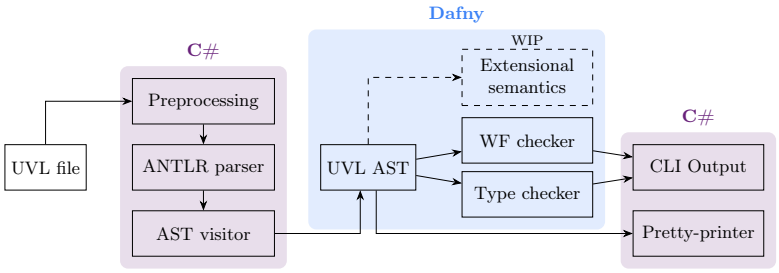

Figure 1. Overall architecture of `mech-uvl`

The C# layer provides the executable front-end of `mech-uvl`. It is built as a .NET command-line interface and can be invoked with `mechuvl`. This command acts as a wrapper around `dotnet run` with the project file. This interface exposes three main commands that use different parts of the toolkit: (i) the `preprocess` command normalizes a UVL file before parsing; (ii) the `parse` command performs preprocessing and additionally translates the file into the abstract syntax tree defined in Dafny; and finally (iii) the `fmt` command performs preprocessing and parsing, then pretty-prints the resulting model. The formatter supports two indentation options: `--indent-size n`, which prints indentation using `n` as a fixed number of spaces, and `--indent-style tabs`, which uses tab characters instead. For example, the following command parses a UVL model and formats the output using indentation.

```
mechuvl fmt --indent-style tabs path/to/model.uvl
```

Internally, the parsing phase relies on the official ANTLR grammar for UVL.[1] Since UVL is indentation-sensitive, the preprocessing normalizes the input by removing comments, normalizing line endings, and inserting explicit indentation. Then, a C# visitor traverses the ANTLR parse tree and constructs

[1] `https://github.com/Universal-Variability-Language/uvl-parser`

the UVL abstract syntax defined in Dafny. The Dafny layer provides the formal core of `mech-uvl`. It defines the UVL abstract syntax together with reusable semantic components for checking and reasoning about models. The well-formedness checker specifies basic structural properties of UVL models, such as valid references, valid cardinalities, uniqueness of feature identifiers, uniqueness of attribute names, and constraints on import aliases. The type checker describes how references, expressions, constraints, attributes, and aggregate expressions are classified. Additional modules address language-level declarations, inference of supported UVL levels, and import environments. While the first step is to formally characterize what it means for such models to be well-formed, the Dafny layer is also intended to provide an extensional semantics for UVL models. This part of the work is currently in progress. Finally, C# manages outputs with a pretty-printer to render the Dafny feature model in canonical UVL form, and a module to report checker results.

3 Design Methodology

Choosing the Formal Verification Tool. Our goal is two-fold: to define semantics and properties and reason about them, and to define executable programs related to them. Interactive theorem provers such as Rocq, Lean, Isabelle/HOL, or PVS are highly expressive and well suited to the first sub-goal. Code extraction is often possible, so programs can also be defined and verified with these tools. However, their learning curve is steep. Tools such as Why3 [12] and Dafny [20] were initially designed as program verifiers. Yet they are expressive enough to specify and reason about non-executable properties and semantics [17,23], relate them to executable programs, and can therefore be viewed as proof environments rather than only deductive program verifiers. Their logic is less expressive than that of proof assistants, but much more automated. Since their languages are smaller and closer to programming languages, they are also easier to learn. Thus, we chose a proof environment rather than an interactive theorem prover, with two candidates: Why3 and Dafny. It is important for `mech-uvl` to be part of the UVL ecosystem, in particular to rely on the existing UVL grammar. Why3 can extract code in several languages, including Java and Python. Why3's Java extraction would have required adapting our development to Java-specific library types, while Python is not a WhyML extraction target. These limitations do not apply to OCaml [24] extraction, but there is no OCaml backend for ANTLR. Dafny code can be translated to C#, which is supported by ANTLR, hence our final choice.

Sources for the Design of Formal Semantics. In the best cases, the artefact to formalize has a reference manual, a standard, or a reference implementation. These may be ambiguous, since they are written in natural language, or contain bugs, but they are usually exhaustive. This is not yet the case for UVL. The reference grammar exists and strongly guides the design of an abstract syntax. We regard the paper by Benavides et al. [4] as the reference on UVL. It is

more pleasant to read than a standard, but not exhaustive enough from a formalization perspective. Some details are provided by other papers, such as [31]. The public UVL model repository of the UVL GitHub organization and the larger UVLHub [27] provide usage examples that can reveal semantic questions and motivate future experiments. Although there is no reference implementation beyond the grammar, several tools exist, e.g. [5,19,21,31], and provide examples of how UVL models may be interpreted semantically.

Principles for Design Choices. The choices we made were guided by the following principles. First, "UVL is designed using different language levels and includes extension mechanisms" [4], so we aimed at making `mech-uvl` as flexible as possible so it fits different understandings of UVL. Second, we aimed at making this first iteration of `mech-uvl` as simple as possible. There is a tension between these principles: including a lot of variability points in our formalization of UVL would make it very flexible but not simple.

Some Design Choices. We made some design choices which seemed quite natural to us, but we do not claim they are the right choices and future iterations of `mech-uvl` may relax some based on the community's feedback.

A first choice concerns constraints local to attributes. As compound record-like attributes can be built from attributes that may contain constraints, constraints may be nested in the inner parts of such a compound attribute, e.g.:

```
features
    Integer Root { MetaData { Name 'Example', constraint !(Root == 0),
    Bounds { constraints [0 <= Root, Root <= 100] },  Middle {
    constraint !(Root == 50) } }, Vector ['0', '1', '+dev'] }
```

This complicates the formalization and is not very readable; thus, we chose to allow local constraints only at the highest level of an attribute declaration.

The UVL grammar allows writing the reference `Root.Metadata.Name` in a constraint. But there is no grammatical construct that allows access to the cells of `Vector`. Also, we found no such reference in the inspected examples. Therefore, we chose to consider that a compound or vector attribute can only be considered as a whole and its components cannot be accessed.

UVL offers types for features and attributes. A question is how to interpret these types in the formalization. There is no ambiguity for `Boolean`. For `Integer`, we could consider machine, thus bounded, integers or mathematical integers. As UVL is a modelling language and not a programming language, we chose the latter. In coherence with the choice on integers, we decided to interpret `Real` as mathematical real numbers not floating-point numbers. These choices do have an impact on the semantics, e.g. constraints such as `X == (X + 1) - 1` and `0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 0.8` may have different meanings depending on whether we consider bounded integers and floating-point numbers or mathematical numbers. Note that the authors of [4] write Float instead of Real. Finally, `String` is interpreted as the Dafny `string` type, i.e. sequences of UTF-16 code units.

```
features
    Semantics
        mandatory
            DerivedAttributes {abstract}
                alternative
                    Strict
                    LocalDerived
                        mandatory
                            LocalDefinitionScope {abstract}
                                alternative
                                    LocalRefsOnly
                                    NonLocalRefsAllowed
                    GlobalDerived
            TypedFeaturesInBooleanPosition {abstract}
                alternative
                    Disallowed
                    AllowedAsPresence
            UndefinedConstraint {abstract}
                alternative
                    UndefinedAsFalse
                    UndefinedAsTrue
            ConstraintEval {abstract}
                alternative
                    Eager
                    ShortCircuit
constraints
    UndefinedAsFalse & ShortCircuit => AllowedAsPresence
```

Figure 2. Variability model of `mech-uvl`'s semantics

It is possible to parametrize the formalization by these types and delay the interpretation only when it is needed, basically when constraint evaluation is needed. However, this introduces quite strong architectural constraints in the DAFNY code, which we preferred to avoid for the first iteration of `mech-uvl`.

Variability in UVL Semantics. There are two main aspects of UVL for which we think it is worth introducing some flexibility, hence some variability in `mech-uvl`. Both are related to type checking feature models. One is related to typed features and constraint evaluation, the other one is related to attributes. In [4], attribute declarations are presented as declaring a key and a value, this value giving a type to the declared attribute. When the value is missing, it is implicitly `true`.

For checking that a constraint is well-typed, one can think that it is enough to have the declared types of features and the declared types (via their values) of attributes. There are, however, cases, not mentioned in [4] but present in inspected public examples, where it is not so simple.

In Boolean feature models, the absence of an optional feature means its value is `false`. For typed features or attributes, either we consider default values (that

is the choice of Damiani et al. in [10] for attributes) or we consider that if a reference is undefined then the whole constraint containing this reference is undefined too. Now, should an undefined constraint be considered true or false? We think both choices are valid, although they are related to other choices.

It seems equally reasonable to either allow any typed feature to be used as a Boolean value, to check whether this feature is selected, or to adhere to strict typing and forbid such a usage. In the former case, this allows guarding constraints involving optional typed features or attributes of optional features. For this style of modelling, it seems natural to consider that logical connectors follow a short circuit evaluation (the alternative being eager evaluation) and that the user does not want to consider constraints that cannot be evaluated because typed features or attributes are undefined. In this context, to avoid unguarded constraints, considering undefined constraints as false seems the appropriate choice. In the latter case, one may still need to express constraints on optional typed features and their values or attributes of optional features. As they cannot be guarded, considering undefined constraints as true, therefore ignoring them, seems the right choice. Assuming for example we have:

```
features
  Root
    optional
      Integer X { Min 10 }
```

in the former case, one would write `X => X >= X.Min` and expect this constraint to type-check, while in the latter case, one would write `X >= X.Min` and would consider the alternative constraint ill-typed. In both cases, the modeller would expect the configurations of this model to have values greater than or equal to 10 for feature `X`. In Figure 2, this variability is modelled by the features `ConstraintEval`, `TypedFeaturesInBooleanPosition`, and the feature `UndefinedConstraint`, as well as by the constraint.

Instead of declaring attributes with a value, some public examples introduce attributes through constraints. The most frequent pattern is a constraint in a local attribute of feature `F` such as `F.TotalPrice == sum(Price)`.[2] It appears to be a valid way of writing a UVL feature model, but it makes type checking more complicated as it actually requires type inference as explained in the next section. We chose to consider the following possibilities, also modelled in Figure 2:

- Introducing attributes in this way is not allowed,
- Introducing an attribute of a feature `F` is allowed only in a constraint local to `F`,
- Introducing an attribute is only allowed in local constraints,
- Introducing an attribute is allowed in any constraint of the current model.

There is no restriction on the form of the constraint. For example, the following constraint introduces the attribute `X` of feature `F`: `0 <= F.X && F.X <= 10`.

[2] Actually it is often `Price == sum(Price)` which we do not expect to be a satisfiable constraint, but this flaw can be solved either as we just did, or by using an optional scope to the aggregate function to exclude the total price from the summation.

In the remainder of the paper, we call "semantics variant" a configuration of the feature model `Semantics`.

4 Well-formedness, Type Checking and Type Inference

To be given a semantics, a UVL model should respect three kinds of properties. It should satisfy core well-formedness conditions, it should be well-typed, and it should respect the language level specification. The first set of conditions expresses that the model is structurally meaningful and that every name is used in a syntactically appropriate role. The second aspect verifies that references, which are typed, are used according to these types. When the goal is to write proofs, such properties do not need to be stated as executable code, or if they are, not necessarily as efficient code. However, to be able to interact with the UVL community as a whole and not a subset interested in proofs, `mech-uvl` provides reasonably efficient executable checkers for the three aspects.

4.1 Well-formedness Conditions and Checker

There are basic properties expected from a model that are not ensured by the grammar:

- **Root feature presence.** Each model must have a root. This is a real WF condition because the grammar admits a model without a `features` section.
- **Valid cardinalities.** Every finite cardinality interval must satisfy $lower \leq upper$. The grammar accepts the shape of a cardinality but does not rule out incoherent intervals such as `[2..1]`.

Then several well-formedness conditions are related to references. In particular, they require every reference occurrence in the AST to be valid for the kind of construct in which it appears. This matters because the same path may be acceptable in one place and unacceptable in another. For example, `pl.MobileApp` may be a valid occurrence inside a constraint if `pl` denotes an imported model and `MobileApp` resolves to a feature in that model. The same path is not automatically valid when it appears as the name of a feature node in the feature tree. In that position, it is acceptable only if it denotes the root of the imported model, such as `pl.Toolkit` when `Toolkit` is that imported root.

We consider the following core well-formedness conditions on references:

- **Valid references.** Each reference is non-empty and has no empty component. This is guaranteed by the grammar for parsed UVL, so it only exists as a specification property, but is not executed by the core checker.
- **Unique feature identifiers.** Declared feature identifiers in one feature tree must be pairwise distinct.
- **Valid feature-tree names.** A feature-tree name must either be a local one-component feature name or a valid imported root attachment. This is already model-environment dependent.

- **Valid reference occurrences.** Every ordinary reference occurrence in constraints and expressions must either resolve to an existing feature/attribute or be an introduced attribute under the selected semantic variant. For a locally declared feature, it should be a unique identifier; for a feature of an imported model, the feature name qualified by the model reference, either an alias or a relative path; and for an attribute, the attribute key qualified by its feature owner unless used in aggregate functions.
- **Unique direct plain attribute names per feature.** A feature must not declare the same plain value-attribute key twice directly on that feature. This is separate from duplicate fields inside nested record values.
- **Valid record attributes.** Record-valued attributes must not contain duplicate field names at the same record level. The condition is recursive through nested records and vectors.
- **Valid aggregate uses.** Aggregate targets for `sum` and `avg` must be bare attribute keys, and optional aggregate scopes must resolve to features. Unary aggregate forms such as `len`, `floor`, and `ceil` require valid reference occurrences.

The last UVL-related checks concern imports. They are checked while building a *model environment*, i.e. a map from optional model key (absent key means the root model), the key being either the alias or relative path of the imported model. Building this environment may return an error when there is a cycle in the imports or an imported model is missing. The termination of this recursive function is proved, and if it does return an environment, it has the import closure and acyclic import graph properties. The properties we consider for imports are:

- **Unique explicit import aliases.** Two imports in the same model must not declare the same explicit alias.
- **Prefix-free visible import qualifiers.** Visible import qualifiers, aliases when present and import paths otherwise, must be prefix-free. This prevents ambiguous imported reference resolution. For example, if we have

```
imports
    lib
    lib.extra
```

  then `lib.extra.A` could be either the attribute `A` of feature `extra` in model `lib` or the feature `A` in model `lib.extra`.
- **Import closure.** Every import in a model in the model environment must point to a model that is present in the same environment. This is an environment property, not a property of one isolated model.
- **Acyclic import graph.** The import graph must be acyclic. This prevents recursive model environments and keeps later resolution and semantics structurally well-founded.

The core well-formedness conditions are written as pure Dafny predicates. These predicates are the specification: they are the properties assumed by later

verified developments, including the semantics. The executable checker is a separate algorithmic layer. It traverses the model environment and returns either `None`, meaning that no core error was found, or the first error that explains why checking failed. The link between both layers is expressed by DAFNY postconditions: if the executable core checker returns `None`, then the model environment satisfies the core well-formedness predicate for the selected semantic variant.

4.2 Type Checking and Type Inference

If the `Strict` and `Disallowed` features of `mech-uvl`'s variability model (Figure 2) are selected, then `mech-uvl` performs a strict type checking. Features and attributes are given types based on their declarations (type declarations for features, indirectly via value declaration for attributes) and all uses in constraints should respect these types. We are not as strict as some languages as we promote `Integer` values to `Real` values if needed. If `AllowedAsPresence` is selected instead of `Disallowed`, then features have two types: their declared type and `Boolean`.

If a child other than `Strict` is selected, it means attributes can be used without being declared. Thus, there is no value to give them a type. We therefore need to *infer* a type for such attributes, based on their usage. `mech-uvl`'s inference mechanism relies on a specific verified finite domain constraint solver. The domain is simply the set of basic UVL types. Each introduced attribute is initially associated with the whole domain, and initial constraints for the constraint solver are built from the usage of the attribute in UVL constraints. Then, a simple propagation algorithm tries to reduce the domains associated with attributes. There are three possible outcomes. First, an attribute may become associated with an empty domain, in which case there is no valid type assignment and the feature model is ill-typed. Second, all the domains are reduced to singletons, in which case the feature model is well-typed. Third, the solver reaches a fix-point, but some domains are still not singletons: the feature model is not ill-typed, but it is under-specified and there are several possible valid type assignments to attributes. There are two sub-cases: really under-specified models are rejected. If the domain is {`Integer`, `Real`}, then, since we formalize these two types as mathematical numbers, it would be sound to choose `Real` systematically. However, this is not what we do. Instead, the domain is concretized according to hints. If the attribute is always an argument of operations or comparisons where the other argument has type `Integer`, then it is assigned type `Integer`, otherwise, it is assigned type `Real`.

The finite domain solver is specified independently of the type checker: a solved assignment satisfies the generated type constraints, and a contradiction means that these constraints have no solution over the initial domains. The inferred types are then used by the executable type checker, whose postcondition connects a call returning no error to the predicate defining well-typed model environments.

4.3 Language Level Checking and Inference

`mech-uvl` checks that if a model specifies language levels then it respects this specification. It does that by first computing the minimal level specification for the model and then comparing it to the specified levels. Due to this design, in addition to checking level specifications, `mech-uvl` can infer a minimal level specification for a feature model.

5 Related Work

Like `mech-uvl`, existing work separates the abstract meaning of a variability model from any particular analysis backend. In our proposal, we clarify the semantics of the rich constructs that are already present in UVL. This leads to UVL-specific choices, such as whether typed features may occur in Boolean positions, whether attributes may be introduced by constraints, and how constraints mentioning undefined values should be interpreted. The main difference is that `mech-uvl` aims to propose a mechanized semantics for a particular language (UVL), while other work has attempted to assign mathematical meanings to feature diagrams more generally. Batory showed how feature models can be translated to grammars and propositional formulae [3], while Schobbens et al. proposed a generic semantics intended to cover several variants of feature diagrams [29]. Cardinality-based feature models have also been formalized directly [9] and through formal-language semantics [28]. However, similarly to existing work, the current proposal addresses attributed and typed feature models. Damiani et al. [10] distinguish logical and extensional characterizations of attributed feature models while Cordy et al. [8] extend product-line model checking beyond Boolean features by considering attributes and multi-features. Clafer [18] integrates variability, structure, and constraints in a language with a formal semantic foundation.

Other approaches encode feature models into specialized semantic domains. Andres et al. develop an algebraic framework for software product lines [1]; Wang et al. use OWL to verify feature models [32]; and Asirelli et al. provide a formal description of variability in product families [2]. Such encodings are useful when the goal is to reuse an existing solver, logic, or modelling environment. `mech-uvl` is complementary: it is intended to provide a semantics before such an encoding.

There is also work on mechanizing product-line reasoning. Janota and Kiniry formalized reasoning about feature models in higher-order logic [16], and Castro et al. mechanized a framework of software product-line analyses in PVS [6,25]. Contrary to our proposal, the PVS-based framework focuses on analyses over software product lines, whereas `mech-uvl` focuses on the language definition, static semantics, and extensional semantics of UVL itself. Contrary to these works, `mech-uvl` relies on proof-oriented definitions with executable code and the existing and maintained ANTLR grammar of UVL. This makes the formalization usable as an experimental toolkit for real UVL files.

Finally, additional work has proposed support for handling UVL files. UVL-Parser [31] extends parser support with language levels and conversion strategies,

and other work studies transformations between UVL and industrial tooling such as pure::variants [26]. Rather than translating models from other notations, recent work builds on models available in UVLHub [27] to generate UVL models for stress-testing tools [30].

UVL-oriented tools have been designed to support editing, parsing, conversion, and analysis. FLAMA [13] provides a modular architecture for automated feature-model analysis; flamapy.ide brings UVL editing and analysis to the web [5]; UVLS implements Language Server Protocol support for UVL [21]; UVL.js explores the JavaScript ecosystem [19]; Strong4VM converts variability models to DIMACS to compute strong transitive dependency and conflict graphs [14]; and variability.dev aims at an online toolbox for feature modelling [15]. These tools are primarily user-facing or analysis-facing. However, each of these tools has made different semantic choices, while `mech-uvl` aims at making these choices explicit in a single semantics-oriented tool. This provides a common formal setting in which UVL analyses and conversions can later be compared.

6 Discussion

Current Limitations. Besides design choices discussed in Section 3 that somehow limit the flexibility of `mech-uvl`, there are some limitations in `mech-uvl` due either to the current formalization architecture or the absence, to our knowledge, of clear directions for making design choices.

The variability model of `mech-uvl` allows attributes to be introduced by constraints rather than declarations, with various choices regarding where such constraints are allowed to appear. One choice that is not present yet, and that would require some refactoring of the current development, is the possibility for a model to introduce attributes in an imported model. We found none in inspected public examples. Thus, unless the community considers this an interesting functionality, we do not plan to add it in the near future.

UVL offers a `namespace` keyword, followed by a reference. While some public examples do use this aspect of UVL, they are single model examples. As the UVL reference paper does not elaborate on this concept, and we do not have any example that uses namespaces in conjunction with imports, we preferred to avoid making a choice on the interaction of namespaces with imports. Therefore, for the moment, namespaces are ignored by `mech-uvl`.

Towards an Extensional Semantics. The extensional semantics is organized in two main parts. The first part, already implemented, defines the structural interpretation of configurations: it determines which concrete feature occurrences may be selected, how cardinalities and groups constrain them, and how imported models contribute to the semantics. The second part, which is work-in-progress, will define semantic evaluation: reference denotation, expression and aggregate evaluation, and constraint satisfaction under a given semantic variant.

The structural part is defined only for checked model environments. Its precondition requires both well-formedness and well-typedness. Feature occurrences

are paths whose components combine a feature identifier with a strictly positive cardinality index. Erasing the indices yields the sequence of feature names along the occurrence path. The structural semantics then checks that each selected occurrence follows declarations available in the composed feature tree, including declarations contributed by imported models. A configuration records three finite components: the set of selected feature occurrences, a partial map assigning values to selected typed-feature occurrences, and a partial map assigning values to attributes of selected occurrences. The implemented structural layer checks that each selected occurrence respects the cardinalities and the tree structure of the feature model. Imported models are handled structurally by treating an imported root attachment as an anchor into the composed configuration. When such an occurrence is selected, the intended semantics projects the global configuration to the suffix rooted at that occurrence and checks the imported model on that projected configuration. The projection operation is implemented, but its recursive use is work-in-progress.

The evaluation part will be added on top of this structural basis. Expression, aggregate, and constraint evaluation will be defined over reference denotations and parameterized by the semantic variant chosen by the user. The expected result is a full extensional semantics as a set of configurations satisfying both the structural obligations and the evaluated constraints, while keeping well-formedness and well-typedness concerns separated from semantic interpretation.

We plan to have the following semantics for constraints when cardinalities are used. Considering a constraint `A => B.X > 0`:

- if `A` and `B` belong to the same cardinality context then there is "pointwise" duplication of the constraint: $\texttt{A}_i$ `=>` $\texttt{B}_i$`.X > 0` with $i \in \{1..n\}$,
- if `A` and `B` belong to different cardinality contexts then the constraint becomes: `(`$\texttt{A}_1$ `||...||` $\texttt{A}_n$`) => (`$\texttt{B}_1$`.X > 0 ||...||` $\texttt{B}_m$`.X > 0 )`.

On the Semantics of Declared Attributes. At least one tool and a feature model in UVLHub consider attribute values as default values that may be overridden in a configuration. Other models and the UVL paper are not explicitly about this aspect but we understand an attribute declaration `A`v, for a value v in a feature `F`, as equivalent to a constraint `F.A ==` v.

On one hand, if attribute values are considered as default values, then there is no need for attributes introduced by constraints, and we could pair strict typing without inference with default attribute values. On the other hand, if attribute declarations are equivalent to an equality constraint then it is not necessary for configurations to give values to attributes, unless introduced attributes are allowed. Rather than a fixed design choice, it seems this could be a new feature in the variability model of `mech-uvl`.

7 Conclusion and Future Work

We presented `mech-uvl`, a mechanized semantic toolkit for UVL. The toolkit combines a C# front-end for preprocessing, parsing, and pretty-printing UVL

models with a Dafny formal core defining a shared abstract syntax, well-formedness conditions, type checking, and type inference. The executable part of this Dafny core is compiled and called by the C# front-end, so that the formal definitions are directly connected to the processing of concrete UVL files. We also identified semantic variation points in UVL, in particular around typed features, undefined constraints, and attributes introduced through constraints. By making these choices explicit and executable, `mech-uvl` provides both a practical basis for checking existing UVL models and a formal basis for further semantic developments.

Ongoing and future work includes completing the mechanized extensional semantics for UVL. Our goal is that the extensional semantics interprets the abstract syntax at the highest UVL language level directly as a predicate relating a configuration and a UVL model (possibly a composed model using imports), and then as Dafny infinite sets of configurations (or finite sets for some UVL levels). The extensional semantics can then be the basis of several research directions including reasoning about conversions [31] and their semantic preservation (or not). We also plan to have intensional semantics of fragments of UVL, e.g. translating UVL to logical formulae, also equipped with an extensional semantics, and to show that the logical transformation of UVL models is semantics-preserving. The executable part of `mech-uvl` is for now designed as a way for the community to experiment with the semantic design choices. As discussed in the previous section, this semantics-focused point of view makes the tool less user-friendly than it could be. If we instead think about it as a linter, some refactoring is needed, for example to better locate the errors, to report several errors rather than the first, and to introduce severity levels for errors and the ability to consider low-severity errors as warnings.

A further direction is to study existing UVL tools, independently of `mech-uvl`, to make their semantic assumptions explicit. Such a study would compare how current tools handle semantic choices, such as attributes introduced through constraints, typed features in Boolean positions, and undefined constraints. This would clarify the de facto semantics already present in the UVL ecosystem.

A complementary direction is to use `mech-uvl` for a systematic exploration of public UVL model repositories, such as the UVL GitHub repository and UVLHub, as well as archived software ecosystems [11]. Since `mech-uvl` makes semantic variants explicit and executable, it can be used to evaluate which variants are actually exercised by existing models and which choices affect their acceptance, typing, or interpretation. Such an experiment would provide empirical feedback to support the future evolution of the UVL language.

Acknowledgements. This work was funded in part by the research federation ICVL through the UniVerSE project: Unification and Verification of Configuration SpacEs, led by Jolan Philippe.

References

1. Andrés, C., Camacho, C., Llana, L.: A formal framework for software product lines. Information and Software Technology **55**(11), 1925–1947 (2013). https://doi.org/10.1016/j.infsof.2013.05.005
2. Asirelli, P., ter Beek, M.H., Gnesi, S., Fantechi, A.: Formal description of variability in product families. In: Software Product Lines (SPLC). pp. 130–139. IEEE Computer Society (2011). https://doi.org/10.1109/SPLC.2011.34
3. Batory, D.: Feature models, grammars, and propositional formulas. In: Software Product Lines (SPLC). LNCS, vol. 3714, pp. 7–20. Springer (2005). https://doi.org/10.1007/11554844_3
4. Benavides, D., Sundermann, C., Feichtinger, K., Galindo, J.A., Rabiser, R., Thüm, T.: UVL: Feature modelling with the universal variability language. Journal of Systems and Software **225**, 112326 (2025). https://doi.org/10.1016/j.jss.2024.112326
5. Benitez, F.S., Galindo, J.A., Romero-Organvidez, D., Benavides, D.: UVL web-based editing and analysis with flamapy.ide. In: International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS). pp. 121–125. ACM (2025). https://doi.org/10.1145/3715340.3715436
6. Castro, T.M., Teixeira, L., Alves, V., Apel, S., Cordy, M., Gheyi, R.: A formal framework of software product line analyses. ACM Transactions on Software Engineering and Methodology **30**(3), 34:1–34:37 (2021). https://doi.org/10.1145/3442389
7. Classen, A., Heymans, P., Schobbens, P., Legay, A.: Symbolic model checking of software product lines. In: International Conference on Software Engineering (ICSE). pp. 321–330. ACM (2011). https://doi.org/10.1145/1985793.1985838
8. Cordy, M., Schobbens, P., Heymans, P., Legay, A.: Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In: International Conference on Software Engineering (ICSE). pp. 472–481. IEEE Press (2013). https://doi.org/10.1109/ICSE.2013.6606593
9. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice **10**(1), 7–29 (2005). https://doi.org/10.1002/spip.213
10. Damiani, F., Lienhardt, M., Paolini, L.: On logical and extensional characterizations of attributed feature models. Theoretical Computer Science **912**, 56–80 (2022). https://doi.org/10.1016/j.tcs.2022.01.016
11. Di Cosmo, R., Zacchiroli, S.: Software heritage: Why and how to preserve software source code. In: Proceedings of the 14th International Conference on Digital Preservation (iPRES 2017). pp. 1–10 (2017), https://www.digipres.org/publications/ipres/ipres-2017/papers/software-heritage-why-and-how-to-preserve-software-source-code/
12. Filliâtre, J., Paskevich, A.: Why3 – where programs meet provers. In: European Symposium on Programming (ESOP). LNCS, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
13. Galindo, J.A., Horcas, J.M., Felfernig, A., Fernández-Amorós, D., Benavides, D.: FLAMA: A collaborative effort to build a new framework for the automated analysis of feature models. In: Software Product Lines (SPLC). pp. 16–19. ACM (2023). https://doi.org/10.1145/3579028.3609008
14. Heradio, R., Cambelo, L., Olivero, M.A., Sanchez, J.M., Perez Garcia-Plaza, A., Fernandez-Amoros, D.: Strong4VM: From variability models to strong transitive

dependency and conflict graphs. SoftwareX p. 102663 (2026). https://doi.org/10.1016/j.softx.2026.102663
15. Heß, T., Ostheimer, L., Betz, T., Karrer, S., Schmidt, T.J., Coquet, P., Semmler, S.N., Thüm, T.: variability.dev: Towards an online toolbox for feature modeling. CoRR **abs/2506.09845** (2025). https://doi.org/10.48550/arXiv.2506.09845
16. Janota, M., Kiniry, J.: Reasoning about feature models in higher-order logic. In: Software Product Lines (SPLC). pp. 13–22. IEEE Computer Society (2007). https://doi.org/10.1109/SPLINE.2007.4339251
17. Jeannerod, N., Marché, C., Treinen, R.: A formally verified interpreter for a shell-like programming language. In: Verified Software. Theories, Tools, and Experiments (VSTTE). pp. 1–18. LNCS, Springer (2017). https://doi.org/10.1007/978-3-319-72308-2_1
18. Juodisius, P., Sarkar, A., Mukkamala, R.R., Antkiewicz, M., Czarnecki, K., Wasowski, A.: Clafer: Lightweight modeling of structure, behaviour, and variability. The Art, Science, and Engineering of Programming **3**(1), 2:1–2:62 (2019). https://doi.org/10.22152/programming-journal.org/2019/3/2
19. Lamas, V., Limaylla-Lunarejo, M.I., Luaces, M.R., Romero-Organvidez, D., Galindo, J.A., Benavides, D.: UVL.js: Experiences on using UVL in the JavaScript ecosystem. In: International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS). pp. 112–115. ACM (2025). https://doi.org/10.1145/3715340.3715437
20. Leino, K.R.M.: Program Proofs. MIT Press (2023)
21. Loth, J., Sundermann, C., Schrull, T., Brugger, T., Rieg, F., Thüm, T.: UVLS: A language server protocol for UVL. In: Software Product Lines (SPLC). pp. 43–46. ACM (2023). https://doi.org/10.1145/3579028.3609014
22. Loulergue, F., Philippe, J., Paul, N.: mech-uvl: A mechanized semantic toolkit for UVL. https://doi.org/10.5281/zenodo.21003070
23. Loulergue, F., Tiemtore, M.H.: Semantics of a subset of Puppet 4.8 mechanized in Why3. In: International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLa). LNCS, Springer (2026), to appear
24. Minsky, Y.: OCaml for the masses. Commun. ACM **54**(11), 53–58 (2011). https://doi.org/10.1145/2018396.2018413
25. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: International Conference on Automated Deduction (CADE). LNCS, vol. 607, pp. 748–752. Springer (1992). https://doi.org/10.1007/3-540-55602-8_217
26. Romano, D., Feichtinger, K., Beuche, D., Ryssel, U., Rabiser, R.: Bridging the gap between academia and industry: Transforming the universal variability language to pure::variants and back. In: Software Product Lines (SPLC). pp. 123–131. ACM (2022). https://doi.org/10.1145/3503229.3547056
27. Romero-Organvidez, D., Galindo, J.A., Sundermann, C., Horcas, J.M., Benavides, D.: UVLHub: A feature model data repository using UVL and open science principles. Journal of Systems and Software **216**, 112150 (2024). https://doi.org/10.1016/j.jss.2024.112150
28. Safilian, A., Maibaum, T., Diskin, Z.: A theoretical framework for cardinality-based feature models: The semantics and computational aspects. Journal of Logical and Algebraic Methods in Programming **97**, 30–54 (2018). https://doi.org/10.1016/j.jlamp.2018.02.002
29. Schobbens, P., Heymans, P., Trigaux, J., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks **51**(2), 456–479 (2007). https://doi.org/10.1016/j.comnet.2006.08.008

30. Sundermann, C., Heß, T., Sundermann, R., Kuiter, E., Krieter, S., Thüm, T.: Generating feature models with UVL's full expressiveness. In: Software Product Lines (SPLC). pp. 61–65. ACM (2024). https://doi.org/10.1145/3646548.3676602
31. Sundermann, C., Vill, S., Thüm, T., Feichtinger, K., Agarwal, P., Rabiser, R., Galindo, J.A., Benavides, D.: UVLParser: Extending UVL with language levels and conversion strategies. In: Software Product Lines (SPLC). pp. 39–42. ACM (2023). https://doi.org/10.1145/3579028.3609013
32. Wang, H.H., Li, Y., Sun, J., Zhang, H., Pan, J.Z.: Verifying feature models using OWL. Journal of Web Semantics **5**(2), 117–129 (2007). https://doi.org/10.1016/j.websem.2006.11.006