Extensions and Scalability Experiments of a Generic Model-Driven Architecture for Variability Model Reasoning

Camilo Correa Restrepo, Jacques Robin, Raul Mazo

Extensions and Scalability Experiments of a Generic Model-Driven Architecture for Variability Model Reasoning

Camilo Correa Restrepo*
camilo.correa-restrepo@univ-paris1.fr
Université Paris 1 Panthéon-Sorbonne
Paris, France

Jacques Robin†
jacques.robin@esiea.fr
École Supérieure d'Informatique Électronique Automatique
Paris, France

Raul Mazo
raul.mazo@ensta-bretagne.fr
Lab-STICC, ENSTA-Bretagne
Brest, France

ABSTRACT

Until recently, the state-of-the-art of *Software Product Line (SPL)* configuration and verification automation consisted of a collection of *ad-hoc* approaches tightly coupling a single input *Variability Modeling Language (VML)* with a single *constraint solver*. To remedy this situation, a novel generic *model-driven architecture* was then proposed that enables using a variety of VMLs and solvers. The key ideas of this proposal were *(a)* the use of a standard logical language (CLIF) as a pivot between VMLs and solvers, and *(b)* the use of a standard data exchange format (JSON) to explicilty and declaratively specify the abstract syntax and semantics of the VMLs to be used in an SPL engineering project and the automated reasoning task to be performed by the solvers.

In this article, we overcome the limitations of this initial proposal in three key ways: (1) we add the ability to reason on textual or hybrid VMLs, rather than only on diagrammatic VMLs, enhancing the versatility of the architecture on the input side; (2) we enable the use of solvers from a third paradigm, enhancing the versatility of the architecture on the output side; and, (3) we present the results of scalability performance experiments of an implementation of this architecture. These results have been achieved without significantly altering the architecture, demonstrating its agnosticism with respect to specific VMLs and solvers. It also shows that it can underlie the implementation of practical variability reasoning tools that scale up to real sized variability model analysis and configuration needs.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; **Software architectures**; • **Computing methodologies** → **Model verification and validation**.

*Also with École Supérieure d'Informatique Électronique Automatique.
†Also with Université Paris 1 Panthéon-Sorbonne.

MODELS '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0504-5/24/09
https://doi.org/10.1145/3640310.3674090

KEYWORDS

Software Product Lines, Automated Reasoning, Generic Architecture, Configuration Automation

ACM Reference Format:
Camilo Correa Restrepo, Jacques Robin, and Raul Mazo. 2024. Extensions and Scalability Experiments of a Generic Model-Driven Architecture for Variability Model Reasoning. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3640310.3674090

1 INTRODUCTION

Software Product Line Engineering (SPLE) is a method to manage the development and evolution of large sets of software products that partially share requirements and reusable software assets implementing them [58]. Each software product in such a set is called a *variant*. Taken together, all of these variants form the eponymous ***Software Product Line (SPL)***. The set of requirements and reusable assets included in a variant is termed a *configuration*. The systematic and disciplined management of the *variability* in the inclusion and exclusion of requirements and reusable assets across all variants is what makes SPLE unique.

For such management tasks, SPLE introduces a novel software artifact, named a *variability model*. It encodes the relationships and constraints among the requirements and reusable software assets. It must capture all the necessary business, technical and regulatory constraints that apply to the SPL. This artifact must be verified to ensure its consistency and minimality. A consistent variability model can then support the semi-automated derivation of all and only of *valid* variants by assembling reusable software assets in a way that respects all the constraints in the model.

Despite considerable efforts by the SPLE community [35, 61, 64], there is no standard language to express variability models. Consequently, every SPLE automation tool has its own tool-specific VML. Although they differ syntactically, the majority of VMLs used in SPLE semantically define coherent sets of requirements, usually called ***features***. The first such VML, that remains the baseline reference in SPLE research, is the ***"Simple Feature Model (SFM)"*** [40]. This modeling language, like most subsequent VMLs, has several key expressive capabilities: (a) it organizes requirements into a composition and abstraction hierarchy, where the lowest level ones are tied to the concrete software assets implementing them; (b) it defines whether requirements are mandatory or optional across configurations; (c) provides alternatives for the refinement of higher level (abstract) requirements; and (d) specifies constraints that range *across* the hierarchy such as conditional inclusion or

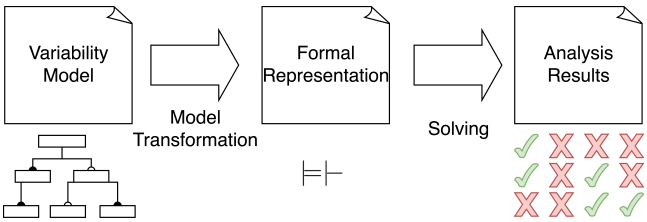

Figure 1: Ad-hoc Automated Analysis Pipeline.

mutual exclusion. Over time many extensions to these models have been defined, such as adding attributes to features [13], having numerical bounds on the number of chosen alternatives [22], or adding constraints that involve more than two features at once [53]. In the literature, they are loosely referred to as ***Extended Feature Models (EFMs)*** when the goal is to distinguish them from SFMs.

Non-trivial SPLs are too large and complex to be manually verified, debugged, or to find valid configurations [48]. In addition, these tasks need to be repeated as the SPL evolves throughout its life-cycle, which may span over multiple decades [16]. Constraint solvers from multiple paradigms have been proposed and tested to automate these tasks [11]. As is the case with VMLs, there is no accepted constraint solver standard for SPLE automation.

As is attested by a recent survey [36] and the comparative analysis presented in [20], the set of current, mature tools used for automated reasoning on variability models follow a general design principle: they directly transform a business-oriented variability model into a formal representation in a solver's input language. A simplified schema of this approach is shown in Figure 1. The reasoning tasks to be performed on the model are then hard-coded into the tools themselves as they need to control the solvers directly to perform them. This has remained true starting from the foundational tool for automated analysis of variability models [40]. Since all these tools either are, or are part of, a textual or graphical variability model editor, they all share the key commonality of being designed for analyzing a specific VML (generally an SFM or EFM [40]) for which they have selected a specific solver.

These approaches do not fulfill the need for ***Domain-Specific VMLs (VML)*** beyond EFMs which is well attested in the literature. In particular, the use of DSVMLs in SPLE projects has both been argued for [17] and their use in combination with or as outright replacements for traditional FMs was explored in [70]. Unfortunately, tooling for a DSVML implies constructing, as is the case with the approaches cited above, an *ad-hoc* model transformation and analysis pipeline that tightly couples the DSVML to the solver.

Product Line Engineering Intelligent Assistant for Defect dEtection and Solving (PLEIADES) [20] departs from these earlier works by proposing a *generic model-driven architecture* that is agnostic to both the input VML and the solver used for reasoning. This proposal subsumes and supports all previous these siloed *ad-hoc* approaches, into a single unifying architectural framework through the use of a standard ***Knowledge Representation Language (KRL)***, the ISO standard ***Common Logic Interchange Format (CLIF)*** [38], acting as a pivot between the VMLs and solvers. To the best of our knowledge, it remains today the sole architecture that can accommodate most past proposals from both the modeling and reasoning sides. Moreover, it demonstrates what should be the way forward for SPLE automation tool engineering, as it shows how efforts expended by each tool development team could all be reused in synergy into one overarching architecture fostering faster overall progress of the field.

The PLEIADES architecture can be said to be a model-driven architecture (MDA) (with respect to the OMG standard MDA [55]) for the following reasons: (a) it adopts the distinction between a Platform Independent Model (PIM), in our case CLIF [38] and (as will be seen later) a generic representation of the constraint problems to be given to the underlying solvers, and the Platform Specific Models (PSM), in our case, the solver specific inputs; (b) it is structured as a pipeline of model transformations from PIM, to PSM, and then to code; and, (c) we model our architecture using the Unified Modeling Language (UML) [63] standard. Furthermore, the language we use as PIM pivot, CLIF [38], while an ISO standard rather than an OMG standard, is the very language used to defined the formal semantics of UML models and MOF metamodel [56], in the OMG Formal UML (fUML) standard [57]. However, we diverge from strict adherence to the MDA approach, by using JSON [21], a W3C standard, rather than MOF, for the meta-models of the VML semantics and SPLE reasoning tasks. This choice is allows for more agilty, a lower barrier of entry for the average developer and more lightweight tooling than a MOF-based approach.

The PLEIADES proposal is accompanied by an open-source prototype in Python that implements the architecture in order to demonstrate its feasibility. Though the original proposal and its prototype are limited in various ways, most notably being restricted to only *Graphical* VMLs, in this article we aim to overcome these fundamental limitations through the introduction of two major extensions to this architecture.

The first of these extensions is to provide the architecture with the ability to reason on purely textual VMLs within PLEIADES. These VMLs form an important part of the landscape of variability models [10]. In this article we explore the implications of such an extension to the original PLEIADES architecture, and how it only leads to minor modifications to its core design. We also examine the problem of extending the original PLEIADES prototype with this functionality. For this, we have chosen an emerging community-led VML named ***Universal Variability Language (UVL)*** [64]. In essence, UVL provides a textual syntax for EFMs and extends it further with concepts from programming languages such as modules and JSON-like nested attributes. This makes variability models using UVL more scalable to large models than graphical EFMs whose editors rarely allow model element reuse across multiple diagrams.

The second important extension is the ability to integrate solvers from other solver families than those originally considered, and in particular, those from the SMT [24] family. With this extension, we also demonstrate how to accommodate a new kind of solver API where the problem instance is constructed directly as set of objects instead of the purely text-generation based approach of previous solver integrations. As will be seen in this article, the modifications that such an addition imply for PLEIADES' core design are, again, minor.

The goal of these extensions is to make PLEIADES demonstrably more versatile both in terms of the input VMLs and the output solver input languages. The ability to reconcile the very different

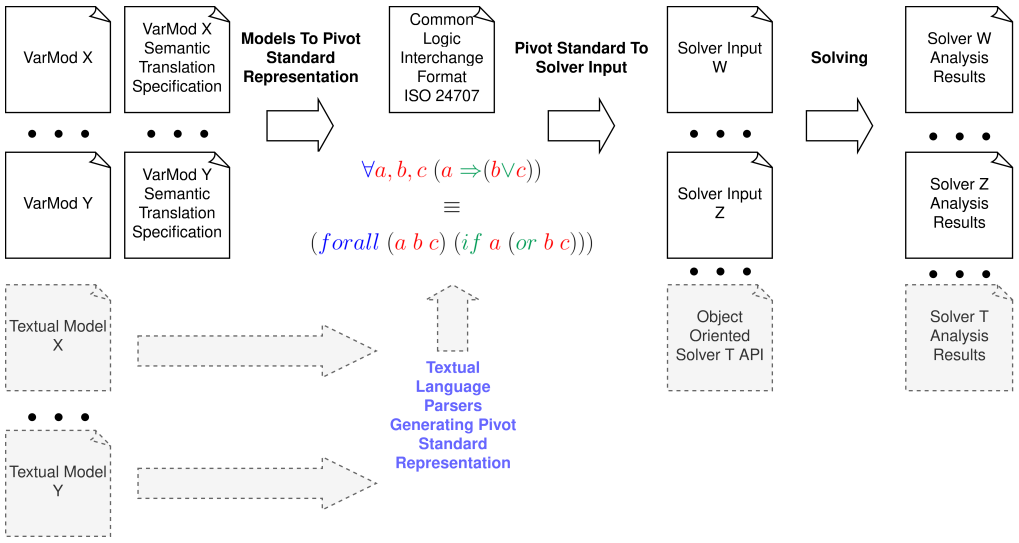

Figure 2: PLEIADES architecture's Analysis Pipeline [20] (white with solid borders) with the proposed extensions (gray with dotted borders).

natures of Graphical and Textual VMLs under one umbrella is a key contribution of these extensions. To show that it can guide the implementation of a tool that is not only versatile but also scales for verifying and configuring large models, we implemented these extensions and carried out scalability benchmarks with this implementation. For this purpose, we used benchmarks VMLs available on the UVLHub[1] repository for UVL models. The main finding is that the PLEIADES prototype, and therefore its architecture, can indeed handle real-sized variability models.

The rest of the paper is organized as follows. In Section 2, we quickly summarize the starting point of our research, the PLEIADES architecture, by explaining how it differs from previous work and discuss what are its limitations towards its stated goal. In Section 3, we then describe the extensions we propose to this architecture and their implementations. In Section 4, we describe the benchmarks that we carried out with this implementation and discuss its results. In Section 5, we compared our work with previous research on providing a SPLE automation tools with some degree of genericity supporting multiple VMLs and multiple solver paradigms. In Section 6, we conclude by summarizing the contributions of the research presented in this paper, identifying its limitations and presenting future lines of research to overcome them.

2 BACKGROUND AND MOTIVATION

2.1 A Proposal for a Generic Model-Driven Architecture for Variability Model Reasoning

The PLEIADES architecture proposed in [20] aimed to provide a generic framework to automate reasoning on graphical variability models. The genericity of the approach hinged on being agnostic to the input VML and to the language accepted as input by the solvers used for automated reasoning. The architecture is itself based on an earlier proposal [18] that proposed utilizing the ***Common Logic Interchange Format (CLIF)*** [38] as the language for encoding VML semantics. The PLEIADES architecture uses CLIF as a pivot language between various VMLs and various solver input languages handling N VMLs and M solver input languages with only $N + M$ transformations rather than $N \times M$ where variability models are directly transformed into formal knowledge bases as done in almost all previous tools reasoning on variability models.

The arguments for CLIF's use is that it was designed to be readable by both humans and machines to represent and exchange first-order logic knowledge bases with some higher order extensions. Its expressive power is superior or equal to that of the languages accepted as input by the four main solver paradigms used to reason on variability models in previous published approaches in the literature:

- ***Logic Programming (LP)*** [46] and its ***Constraint Logic Programming (CLP)*** [29] extensions which were used for the foundational VML analysis tool [40] and the original VariaMos tool [68].
- ***Constraint Satisfaction Problem (CSP)*** solvers and their extensions for ***Constraint Optimization Problems (COP)*** [25] which were used for the COFFEE [69] and Familiar [1] tools.
- ***SATisfiability (SAT)*** solvers [34] and their extensions with ***Satisfiability Modulo Theories (SMT)*** [24] were used for FeatureIDE [66], FlamaPy [32], Splot [50], Glencoe [60], KernelHaven [43], and pure::variants [14].
- ***Description Logic (DL)*** engines and their semantic web reasoning extensions [6] which were used for the AUFM tool [54].

The key insight in [18] is that the semantics of all the graphical VMLs can be expressed in CLIF, which can in turn be used to generate input specific to each solver (or family of solvers) [20]. In this way, adding a new VML or solver would only need a single

[1] https://www.uvlhub.io/

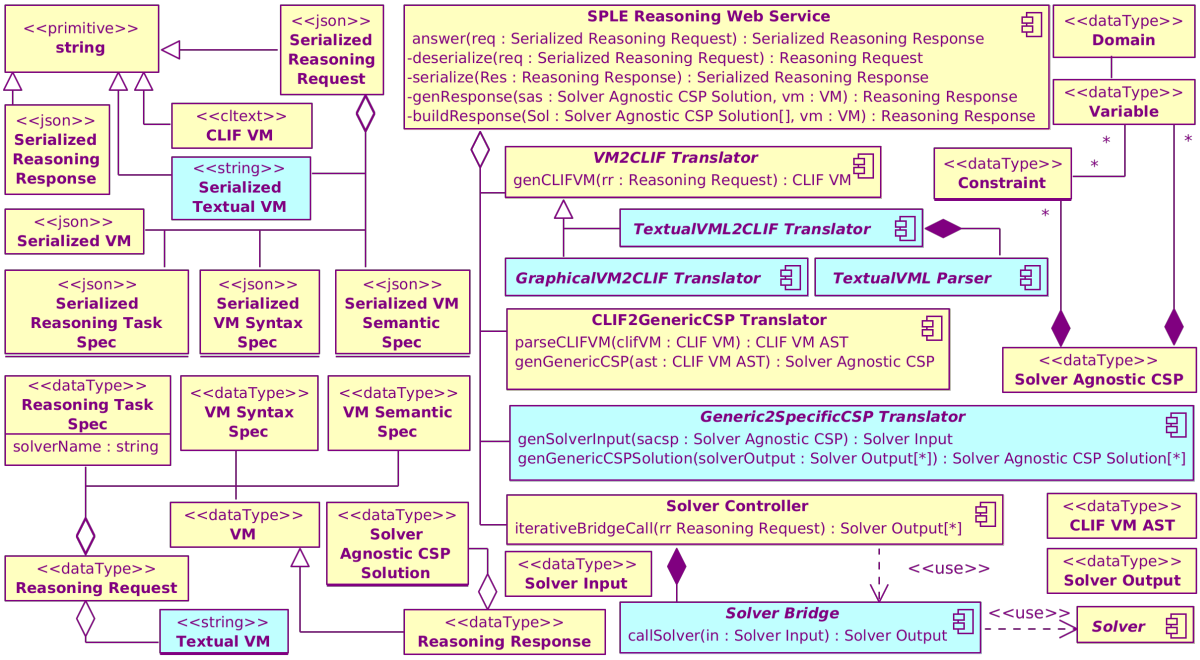

Figure 3: PLEIADES architecture's [20] Component Diagram with the proposed extensions. The elements from the original proposal are in yellow and the modified or added components are in cyan.

additional translation on the VML (input) side or on the solver (output) side, all within the same architecture, while reusing already available language transformations.

A simplified schema of the PLEIADES architecture's pipeline is shown in Figure 2. In this figure, all the elements outlined in the original proposal are in white with solid borders. It essentially consists of expanding the basic pipeline shown in Figure 1 to allow for the many-to-many correspondence of VMLs and solvers by utilizing CLIF as an intermediary. The gray elements with dotted outlines correspond to the extensions necessary to cover the architecture's limitations, which are the subject of Section 2.2.

The PLEIADES architecture is designed as a REST Web Service [59] where the model in a given VML is accompanied by two declarative specifications. The first encodes the semantics of the VML in the form of a mapping from the VML abstract syntactic elements to CLIF sentence elements. As shown in Figure 2, this specification is used to translate the variability model into a CLIF sentence. This sentence can then be translated into a sentence of the KRL understood by the solver chosen to reason on the variability model. The second declarative specification associated with the variability model defines the reasoning task to be performed by the solver. These tasks can be for example detection of a specific class of VML defect or interactive configuration.

The new, extended PLEIADES architecture has five top-level components (c.f. [20] for a much more detailed explanation of each component and its design choices), as shown in Figure 3:

- The entry point web service, `SPLE Reasoning Web Service`, that handles the web requests and orchestrates the operations provided by the other components.
- A component, `VM2CLIF Translator`, that transforms the input variability model into CLIF. In the original PLEIADES architecture, it was a stand-alone component tied to the transformation of the graphical models. As shown in Figure 3 the extended architecture makes it abstract and specializes it into a `Translator` component for each type of model (textual and graphical), as described in Section 3.1.
- A component `CLIF2GenericCSP Translator` that transforms the CLIF model into a semantically-equivalent solver-agnostic (generic) CSP specification. While CLIF can represent the logic underlying any solver input, as they are all ultimately expressible in first-order logic, its concrete syntax differs from that of the solver input languages. This transformation involves parsing the CLIF sentences string into an *Abstract Syntax Tree (AST)* [2] and then translating it into the AST for the solver-agnostic CSP specification language. As will be seen in Section 3.2, this remains true for the addition of the third solver family that we explore in this article.
- A component (`Generic2SpecificCSP Translator`) that takes a solver-agnostic CSP specification and translates it into a solver specific input. It is shown as modified in Figure 3 because it must include an additional translator for the third solver family.
- A component (`Solver Controller`) which interprets the requested reasoning task, orchestrates multiple calls to the solver when needed, assembles the solutions returned by the solver and sends the result to the top-level web service component. It is composed internally of a component (`Solver Bridge`) whose purpose is to handle the life-cycle of the solver, provide its input and extract its raw output. Given

that each solver has its own specific API, one must have set of bridges, one for each solver. However, to preserve the separation of the controller from the solver-specific API, each bridge exposes the same interface to the controller, applying the dependency inversion pattern [49].

This multilayered architecture corresponds to the top portion of the simplified schema of the pipeline depicted in Figure 2. The set of operations and the objects passed through each layer of the architecture are detailed in Figure 4. Of particular note is that only three activities change from the initial to the extended architecture. These changes are encapsulated inside these activities and correspond to the modifications needed for accommodating the textual languages and more solvers. In this case, the nature of the pipeline stays the same. We will return to this in Section 3.3.

2.2 Limitations of the Original PLEIADES Architecture

The *original* PLEIADES architecture suffered, however, from two main limitations: it did not consider neither Textual VMLs and their semantics, nor a larger set of solver families. In addition, the publications about it [18, 20] did not discuss the degree to which the input languages to the different solvers could be generalized and did not include scalability benchmarks.

The first limitation of the original PLEIADES architecture is quite significant since, in practice, there are many relevant purely textual VMLs, as attested by a recent survey [10]. Moreover, each of these languages has its own grammar and structure which contrasts with the principle outlined in the PLEIADES proposal where all Graphical VMLs are Directed Graphs [15]. This VML-agnostic graph structure was leveraged in the original PLEAIDES architecture to generically transform the variability model into CLIF. But textual VMLs lack such a language-agnostic graph structure, which is due to the fact that, although *as a product of parsing* one may generate a graph-like AST [2], text, by itself, is unstructured.

The second limitation of the original PLIEADES architecture is tied to the construction of the input for other sets of solver families than those originally considered, ***Constraint Satisfaction Problem (CSP)*** solvers [25] and ***Constraint Logic Programming (CLP)*** solvers [29]. In the architecture, both of these solvers are managed through the generation of a string representation of the input problem to be sent to the solver, and then invoking a subprocess to read said string and return the result. This differs from, for instance, the API offered by the Z3 solver [23] from the SMT family [24] that offers an object-oriented [42] API to construct the problem programmatically.

The lack of scalability evaluation of the original PLEIADES architecture and prototype implementation was also important since one can suspect that there will be a significant trade-off between the genericity of the architecture and the computational overhead of decomposing the transformation into several intermediate layers.

3 EXTENSIONS TO THE ORIGINAL PLEIADES ARCHITECTURE AND PROTOTYPE

Faced with the limitations evoked in Section 2.2, we have addressed them one by one by proposing and implementing a series of extensions to the original PLEIADES architecture. The purpose of these extensions is twofold. First, we aim to determine, by critically examining the original proposal, whether the architecture is indeed amenable to the integration of these key features without significant alteration as originally claimed. Second, we seek to show how to concretely add these extensions and their impact on the overall architecture.

The starting point for the extended PLEIADES prototype is the open-source original prototype. This is important since one of the key claims in the original paper hinge on extensions such as those presented here being able to be effectively implemented. Moreover, it is claimed as one the main requirements for the original architecture that it must "[... s]upport low-cost [...] addition of new DSVMLs" and that it must "[... support] low-cost addition of interoperability with solvers from different paradigms" [20]. We can only verify these claims in practice by actually performing such additions, and in particular, measure the "cost" by the effort necessary to adapt the architecture and its implementation.

3.1 Handling Textual VMLs

Textual languages pose a challenge to the original PLEIADES architecture because the abstract syntax of each of them is specified as a different grammar, whereas the abstract syntax of graphical models can share a graph-based meta-model. Processing a textual language thus requires a parser [2] that can recognize the language, detect the well-formedness of models, and build a structured representation, the Abstract Syntactic Tree that supports both their programmatic manipulation and the association with its semantic specification. Since the input is raw text, it cannot be trivially put into a structured representation, such as JSON [21] over which the original semantic specification approach could be applied.

This being the case, we posit that there are two possible ways to add support for such languages within the architecture:

- Adding an additional layer to the architecture that transforms the textual model into its equivalent in graph form and then said model is fed to the rest of the architecture.
- Adding a layer *in parallel* to the graph-based CLIF generator that transforms the textual model directly.

We consider that the latter of these options is the most adequate for several reasons. Transforming the model into a graph-based language would necessitate the additional definition of an abstract syntax and translation rules on top of the need for the parser and the code generation rules to create the graph in the first place. This double indirection before even entering the pipeline of the original architecture would be too costly in terms of additional computation. In addition, the presence of relations that are not expressible as edges in a graph (involving several nodes at once) would nevertheless need to be rendered textually [20].

The process of integration involves the addition of only two elements to the architecture *for each new language*:

- A parser that generates the CLIF encoding of the original textual model.
- A mechanism to invoke the parser from the architecture and provide the generated CLIF to the `CLIF2GenericCSP Translator`.

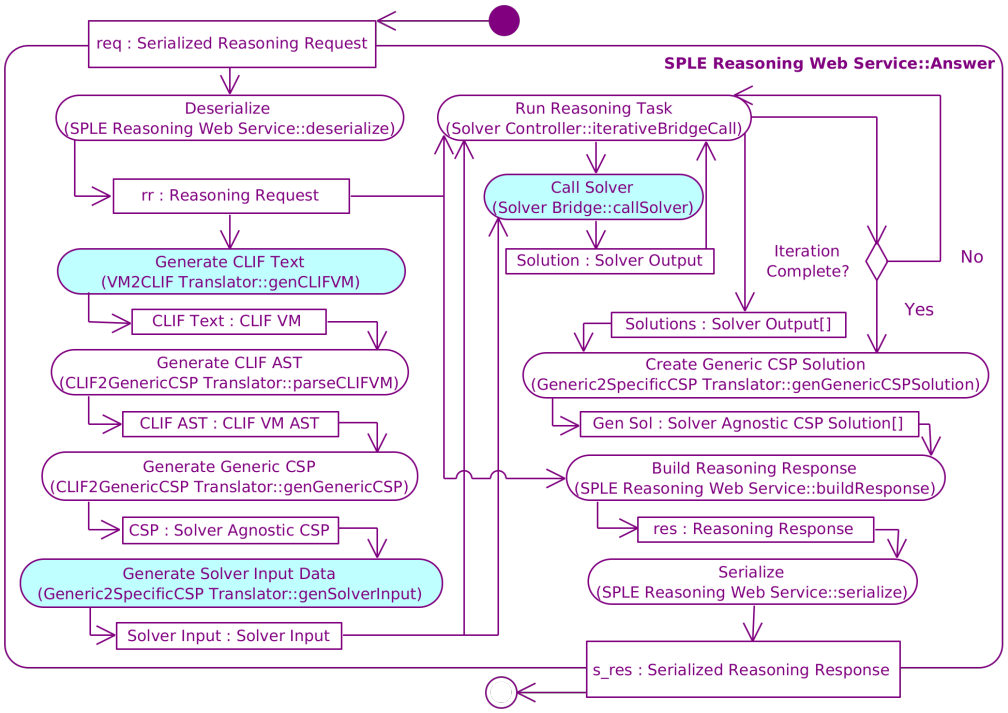

Figure 4: PLEIADES architecture's [20] Analysis Pipeline Activity Diagram with the proposed extensions. The activities and objects from the original proposal are in white and the modified or added components are in cyan.

3.1.1 The Universal Variability Language. Given that textual VMLs are all different syntactically and semantically, it was necessary to make a choice as to which would be the most suitable to concretely realize the extension of the architecture. Ideally one would utilize a "standard" language for this purpose. Unfortunately, while there have been several standardization efforts, most notably the ***Common Variability Language (CVL)*** [35] and the ***Variability Exchange Language (VEL)*** [61], neither of these efforts has led to the publication and adoption of a standard. The latest effort that has managed to garner support in the SPLE community is the ***Universal Variability Langauge (UVL)*** [64]. This language is an attractive choice because it has been adopted as the export format for several popular variability modeling tools such as FeatureIDE [66] and pure::variants [14], which would open up the possibility of interfacing a prototype of the architecture with these tools. An example UVL model for a computer [65] is shown in Listing 1.

3.1.2 Implementing the Extension. Having decided on the use of UVL, now comes the question of its implementation within the extended PLEIADES architecture. The first step is defining its logical semantics in CLIF of the constructs of the langauge. These semantics follow closely those defined for Extended Feature Models [12, 26, 48, 51]. Though no complete description of the complete logical semantics of UVL has yet been presented, we have based our semantics on those hard-coded into two earlier tools that use UVL as their input format: FlamaPy [31] and the UVL Language Server [47] IDE extension.

```
features
    PC
        mandatory
            RAM
                or
                    "8GB"
                    "16GB"
            CPU
            "Power Unit"
                alternative
                    Large
                    Small
        optional
            "Designated GPU"
constraints
    "Designated GPU" => Large
```

Listing 1: Example UVL model for a computer reproduced from [65].

With the semantics in place, it was necessary to construct a parser capable of producing CLIF models based on them. To do this, we have based our implementation on the grammar of the parser proposed in [65], the main reference implementation of UVL. The structure of our parser is based on the classic lexer/parser/code generator structure [2]. Its implementation has been done in Prolog and included in our extended version of the original prototype.

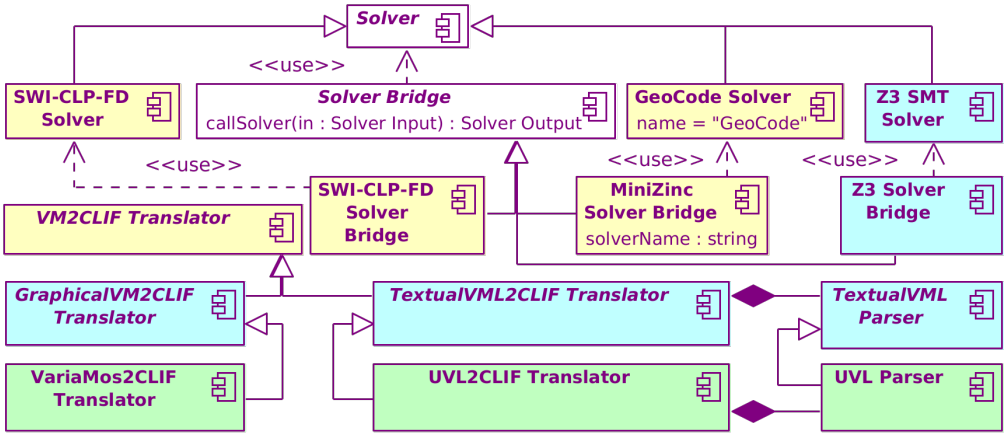

Figure 5: Current Extended Prototype instantiating the PLEIADES architecture with its additions. Components in yellow (concrete) and white (abstract) correspond to components from the original architecture. Components in green (concrete) and cyan (concrete) and blue are the newly implemented components.

We then instantiate the `TextualVM2CLIF Translator` component inside the architecture in Python and use Prolog's Python Foreign Language Interface [71] to call the parser and receive the resulting CLIF.

3.2 Adding solvers from the SMT family

SMT solvers are the natural choice as the next solver family to integrate into the architecture for they have been applied with good results in variability model reasoning (c.f., for example, in [47]), as well as for reasoning tasks in a variety of other domains [24]. Out of the solvers available in this family, we have selected Z3 [23], a mature, robust and well-documented solver with a Python API which should considerably facilitate its integration into PLEIADES.

The problem of extending the architecture to handle an SMT solver raises two main questions which we will address in order:

- *Do solvers from this family fit into the approach of constructing a* `GenericCSP` *as done in the PLEADES architecture?*
- *Is the mechanism for controlling the solver from the architecture analogous to that of the other two solver families?*

3.2.1 Correspondence of the Structure of a Generic CSP with the Problem Structure for an SMT Solver. The expressive power of an SMT solver is dependent precisely on the theories over which it reasons [24]. Considering that there is a direct correspondence between the expressivity of constraint solvers (either CLP or CSP) and SMT solvers equipped with a theory for integer arithmetic and equality [9], it is possible to affirm that, at a minimum, they are capable of expressing the same types of constraints. In all cases we have the same three fundamental elements of a "constraint" problem: a set V of variables, a set D of domains whence the variables take their values (like the natural numbers $\mathbb{N}$ or the integers $\mathbb{Z}$) and a set C of constraints that must be respected [25]. This leads to an affirmative answer to the first question posed, which is whether it is possible to frame an SMT problem under the same `GenericCSP` structure as is done in the architecture.

3.2.2 Controlling an SMT Solver. As described in the original architecture [20], the `Generic2SpecificCSP Translator` generates a representation in the textual input format of a specific solver. While it would be possible to directly control Z3 by running a sub-process and writing the model out in the SMTLib [8] text-based format, this is disadvantageous since it would require including another parser to interpret the textual output and reintroduce it back into the architecture. Instead, Z3 exposes an object-based API within Python to both construct the problem and its solution. The Z3 runtime is handled through Python's C Foreign Language Interface. Moreover, from an extensibility viewpoint, it is interesting to be able to interact with solvers with this type of API (e.g. PySAT [37] that exposes various SAT solvers with a single Python API).

The radical change is that instead of defining a translation from the `GenericCSP` into text, as described in [20], a set of objects is constructed directly within the Python runtime. Running the solver and obtaining the results consists of simply calling the appropriate functions of the Z3 API.

3.3 Changes to the Model-Driven Architecture

The two large extensions to the architecture outlined above entail remarkably few changes to its structure. In particular, only one of the original architecture's high-level components (the `VM2CLIF Translator`) is modified in a fundamental way. This component was proposed as being the sole entry-point for translation of variability models in the orignal PLEIADES architecture, making use of the *declarative semantics specification* to generate the CLIF representation of the model. The introduction of textual models forces the approach to change whereby the concrete component that used to correspond to the `VM2CLIF Translator` becomes a `GraphicalVM2CLIF Translator` as depicted in Figure 3. This accommodates for the possibility of having another type of "Translator" to handle textual languages (shown as `TextualVML2CLIF Translator`). It includes the parser necessary for the analysis of the language and the construction of the CLIF representation. Concerning the generation of the solver input for the new solver family, the

`Generic2SpecificCSP Translator` only changes *internally*, by including the additional translation mechanism. Its place in the architecture remains the same, as does the signature of its operations. In the case of the `Solver Bridge`, the only change to the architecture is that an additional subclass must be added corresponding the new solver, though this is mainly tied to the implementation.

3.4 Prototype Implementation

Given that the architecture changes relatively little, and as detailed in Sections 3.1.2 and 3.2, the implementation of our extended prototype primarily involves the addition of the new components for textual VMLs and the SMT solver. We depict in Figure 5 how the prototype specializes the abstract components of the architecture. Given that the original prototype closely follows the original architecture [20], this is also the case for the new prototype. One of the key motivations behind the implementation is to demonstrate concretely that the architecture's improvements are indeed feasible.

The prototype is made freely available as an open-source tool at https://github.com/ccr185/semantic_translator. The exact release version for this article can be found in [19].

4 BENCHMARKS AND EVALUATION

One of the principal goals of this research is to determine whether the architecture, in its extended form, is indeed capable of scaling as the sizes of the variability models grow. We define the notion of "real-sized" variability models as those ranging into the hundreds of features and constraints, as argued, for instance, in [62]. In addition, since the architecture is ultimately designed to provide tooling within a larger modeling framework, be it graphical or textual, we argue that scalability is the ability of the framework to present results within a reasonable time frame to users. Previous studies on user behavior [5, 33, 52] suggest that users expect to receive results from information systems in the range of 2 to 10 seconds.

4.1 Nature of the Dataset

For our scalability test we have selected the largest dataset available on UVLHub [67], a repository of shared UVL models. In particular, we have picked the "Complete SPLOT Dataset[2]", which contains a conversion into UVL of nearly all models originally created for the SPLOT [50] tool. This dataset is pertinent because it is representative of the large range of models that are encountered both in academia and in industry [7, 30]. The model sizes range from 11 to 625 features, with a similar range of cross-tree constraints among the features. Since the models in the dataset are based on user submissions to SPLOT [50], it cannot be expected that all models are necessarily satisfiable, in particular given the fact that the cross-tree constraints users may place on the models are not checked before inclusion into the original repository nor for the dataset.

4.2 Measurements and Set-up

In order to carry out the measurements for our benchmark, we have first instrumented the prototype implementation to gather statistics on each element of the dataset, in particular the number of features and constraints that are contained in each model. Next, we have prepared a set of scripts that run each model through the prototype and gather the precise runtime (using the system clock) of each run. A repository with the exact scripts and data needed for replication is made available at [19].

The testing procedure is then done as follows for each of the available solver back-ends:

- Launch a local instance of the prototype on the test machine as a web service.
- Create a .csv file that will store the runtime information.
- Find all the *".uvl"* files in the benchmark folder and create a request JSON file including the contents of each model together with a specification of a reasoning task, checking for satisfiability of the model.
- Run a web request to the prototype, gathering the time taken for the entire round-trip without the (very variable) network overhead.
- Record the response from the prototype and the runtime information in the CSV file.

In order to have a very precise view of the runtime of the system, the instrumentation measures the exact time the thread handling the requests spends on the entire pipeline within the prototype. We disregard the network overhead in the measurements for two reasons: (a) it is likely to be highly dependent on the specific web server and network conditions in real deployments, and (b) it generally is not consistent, even across identical requests in our testing. In addition, for efficiency reasons, we ran 16 tests in parallel on the server, each being handled by a server thread. This procedure is ran a total of fifteen times, five times for each of three solvers included in the prototype, which are then averaged to give a final runtime for each model.

The machine on which the tests were carried out has the following characteristics:

- CPU: Intel i9-11900K @ 3.5 GHz (8 cores)
- RAM: 4x16GB DDR4 @ 3200 MHz
- OS: Ubuntu 22.04 LTS
- Python Runtime: Python 3.10.6 (Anaconda)

4.3 Results

The primary run-time results are shown in Figure 6. The figure depicts three data series: (1) the average run-time for each model using the SWI-Prolog solver (shown as green dots), (2) the average run-time using the MiniZinc solver front-end (shown as purple dots), and (3) the average run-time using our newly-added Z3 SMT solver (shown as blue triangles). The figure is semi-logarithmic, with the axis corresponding to the model size being logarithmic. The model size is taken to be the number of features added to the number of constraints (cross-tree or otherwise) contained in the model. We have also fitted exponential curves to the three data sets.

The data suggest that the runtime performance for models in this size range is remarkably similar across the different solver back-ends. Moreover, their performance is nearly identical for models with a total size smaller than 100. Incidentally, it is around this point that the computation time begins to grow exponentially, in line with what one would expect of problems that are fundamentally NP-complete and hence exponential in the worst case [25, 34].

[2]Available at https://www.uvlhub.io/dataset/view/20.

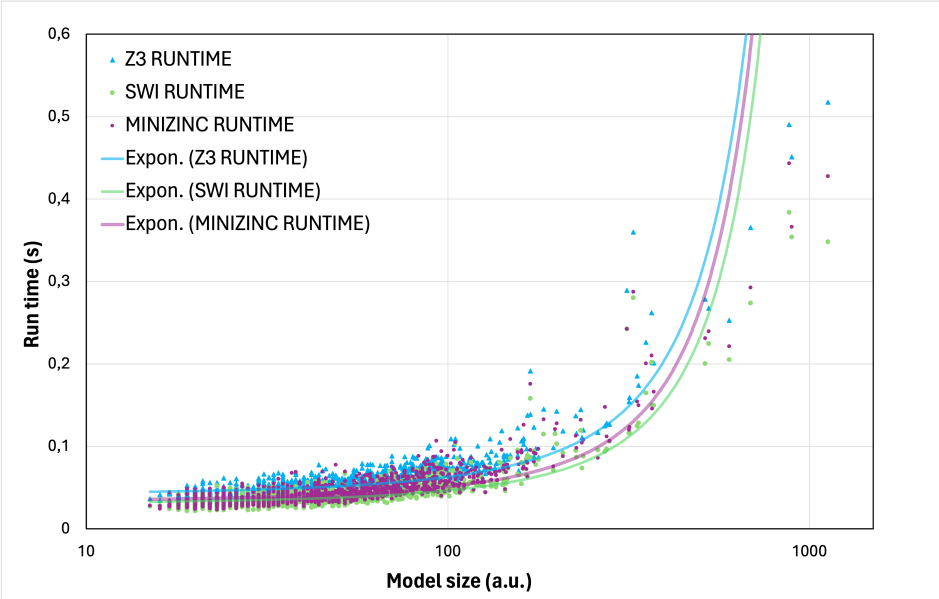

Figure 6: Performance Data for the extended PLEIADES architecture.

This suggests that there is a minimal overhead for the orchestration of the pipeline operation's, notably the transmission of data to/from the solvers and to/from the UVL parser of approximately ~50 milliseconds.

4.4 Discussion

Several conclusions can be drawn from the data regarding the prototype and its implications for the architecture. First, it is clear that there is a minimal overhead that will always be present, independent of any particular prototype, simply because the architecture requires orchestrating the communication between different tools with, in general, their own independent runtime, that requires memory allocation, loading inputs and transmitting the outputs. Nevertheless, for the cases examined, this overhead is minimal and is unlikely to be the cause of future scalability issues.

A second conclusion is that for the entirety of the dataset in question, the execution time of the architecture's prototype conforms to our informal measure of "a reasonable amount of time", i.e., providing an answer in less than 2 seconds. The exponential curves that our modeling software has fit to the data do not correspond exactly to the trend apparent in the upper range of data, since they seem weighted by the numerous models of small and intermediate size. Both the curves and the data suggest an exponential trend as the model size grows, which corresponds to the expectations one would have of combinatorial problems as is the case of determining satisfiability of variability models [45]. The implications for the architecture are twofold: (1) just like for any other automated analysis tool, the eventual bottleneck in computation will be solving the model, whose scaling will depend on the optimizations and meta-heuristics implemented by the solver; and, (2) as larger and larger models are employed, care must be taken to ensure that the processing overhead does not augment significantly, that is, not just the overhead for the orchestration, but the overhead for parsing the CLIF models or the variability models, constructing the Generic CSP, generating the solver's input, and all other miscellaneous tasks.

A third and final conclusion is that the performance of the newly added solver is nearly identical though consistently slower than that of the two earlier solvers. We hypothesize that this may be due to the object-oriented API of Z3, since Python will necessarily need to perform a large amount of object creation and its associated memory management, rather than as a consequence of belonging to the SMT solver family.

It is important to highlight that the prototype we've presented manages to scale well in the range we have tested. This lends credence to the architecture's adequacy as the design for a generic tool for automated analysis of variability models. Though our implementation has been done in Python, in our view, there is nothing that would impede an instance of the architecture from being rebuilt in another language, e.g. C++ or Rust, where the performance could potentially be much improved. In addition, this being a research prototype, further improvements to the implementation can drastically reduce the overhead for the pipeline, for instance, with a high-performance parser for CLIF.

4.4.1 Threats to Validity. Our work is subject to several threats to its internal and external validity. In terms of ***Internal Validity*** we have two main concerns: first, the feasibility of the architecture (a threat also cited in the original proposal [20]), and, second, the validity of the measurements here presented. We counter these concerns in several ways. The prototype we presented corresponds very closely to the architecture's design, whose modifications we examined and justified in Section 3. Moreover, the scalability testing we used to stress-test the prototype, and presented in this article, serves to give faith in the correctness of the implementation and the

architecture's ability to cover its intended purpose. We also show that the PLEIADES architecture, as conjectured in the original proposal, can indeed be extended beyond the graphical modeling environment with which it was built and originally tested. The architecture has been designed with a standard and well-understood modeling language (UML) [63] with feedback from practitioners, peers in the field, and users of the tool.

As for the measurements, we have utilized the Python runtime's `time` module's `thread_time_ns` function to precisely measure the time spent on computation with nanosecond accuracy. We have also performed the measurements directly within the prototype's code *independently* of the network handling to avoid any interference with the measurements, and to isolate the runtime that is dedicated to the processing pipeline. We have performed several runs for each model and solver combination and averaged the results.

Next, in terms of ***external validity***, we firstly recognize that our scalability experiments concern primarily the newly added extension on the input side. We are unfortunately unable to perform such an experiment with graphical models for the principal reason that, to the best of our knowledge, the authors of the VariaMos [68] tool, which was target of the original prototype, do not have a corpora of models available for these purposes. We have also not tested the entirety of the datasets available on UVLHub. We have chosen the SPLOT dataset primarily because of the large number of models it contains and the large diversity in size it presents, and that it is representative of models encountered both in research and practice [7, 30].

Finally, the informal definitions of *real-sized* models and *reasonable amount of time* pose a threat to ***construct validity***. We have sought to base the former on previous empirical studies on industrial models [62], though we are aware of the existence of considerably larger models existing in industrial practice [41]. The latter definition we have based on studies of user-behavior [5, 33, 52] w.r.t to the response time of applications, though we do concede that the exact value is arbitrary to a degree, there being no consensus on a precise value for this parameter.

5 RELATED WORK

In this paper we present an extension, implementation and scalability evaluation of PLEIADES [20], a generic model-driven architecture for variability model verification and configuration automation for models written in a variety of languages and leveraging solvers from different paradigms. In [20], the PLEIADES proposal was compared with previous work in variability model verification and configuration automation, notably Feature IDE [66], its related project FAMILIAR [1], FlamaPy [31], the COFFEE Framework [69], SPLOT [50], Glencoe [60], ClaferTools [4], Kernel Haven [43], pure::variants [14], and Gears [44]. This comparison was done along five dimensions: (a) architecture model, (b) VML expressiveness, (c) solver input language expressiveness, (d) whether that language was a standard and (e) whether the model verification and/or configuration tasks carried out by the solver were declaratively specified as data or hard-wired in imperative code making multiple queries to the solver which are needed for complex tasks. We now summarize these results. Concerning (a), the PLEIADES architecture was the only one, with Kernel Haven's, to cover both structural and behavioral aspects, with PLEIADES being more fine grained (and thus easier to faithfully implement). Concerning (b), PLEIADES was the only one to simultaneously support integer attributes, first-order constraints and runtime, context-aware variability models for dynamic SPLs. Concerning (c), the CLIF language proposed in PLEIADES is the only one, along with Clafer, to simultaneously cover integer domains, first-order constraints, utility functions supporting search for an optimal configuration and incremental reasoning, but with Clafer's meta-programming capability being unclear. Concerning (d), the uniqueness of the proposal to use the CLIF standard was highlighted. Concerning (e), it was found that PLEIADES and Kernel Haven were the only two proposing a declarative specification of the reasoning task (verification or configuration) to be carried out by the solver.

Concerning the PLEIADES extensions that we present in this paper, the closest related approach is FlamaPy's [31]. It supports multiple VMLs, including UVL. However, it supports a single solving paradigm, SAT, whose KRL is known to be far less expressive than those of the CSP, CLP and SMT paradigms incorporated in our PLEIADES extension. These three paradigms can all be viewed as extensions of SAT to concisely represent non-Boolean, first-order and soft constraints. Adding new solver paradigms would require rethinking FlamaPy's architecture with its pivot language at the *abstract syntax* level, in the direction of PLEIADES with a *semantic* level pivot language.

Another recent approach is that of the UVL language server (UVLS) [47]. It suffers from the inherent limitations of an *ad-hoc*, single VML, single solver architecture, for it transforms the UVL model directly into an SMT specification to be solved by Z3 [23].

To the best of our knowledge, no reasoning scalability study has been published for either of those tools. Performing such a study is difficult for a third party, since the user interfaces of those tools do not provide options to generate run time performance logs.

6 CONCLUSION

In this article, we have presented two major extensions to the initial PLEIADES architectural proposal [20] to provide a generic model-driven architecture for automated reasoning on variability models. The first extension, detailed in Section 3.1, concerns the capacity of the architecture to handle purely textual VMLs, whose absence was its main limitation. We showed how to extend the architecture to handle textual VMLs in general. We also showed how to implement this extended architecture to make it capable of reasoning on models written in UVL [64], a textual VML that is starting to gain traction in the SPLE community to help provide some SPLE tool interoperability.

The second extension, presented in Section 3.2, details the extension of the architecture to leverage solvers from a third paradigm, namely SMT solvers[24], complementing the CSP and CLP paradigms already incorporated in the original architecture. We show how to extend the architecture to add SMT solvers in general and how to implement this extension by adding the Z3 SMT solver [23]. For the resulting extended architecture we have provided detailed structural and behavioral models.

The second major contribution presented in this paper is the scalability and performance experiments we carried out on the

prototype implementation of this extended architecture. We have instrumented the extended prototype for it to be able to generate detailed performance measurement logs. We also examined the relative performance of the new SMT solver, Z3, w.r.t. the CSP meta-solver MiniZinc (with the Geode solver) and the CLP solver in SWI-Prolog (with the CLPFD library) already integrated in the implementation of the original architecture. We also discuss the minimum overhead that a generic architecture introduces, as compared to single-VML single-solver tools, in return for its ability to be agnostic and versatile in terms of the input VML and the solver used to reason on it.

It is our hope that this extended architecture, the ease with which it can be instantiated, and its provided open-source baseline implementation will constitute a stepping stone towards the creation of an interoperable, reusable specialized SPLE tool ecosystem dispensing researchers of re-implementing a whole variability model analysis tool from scratch to test any novel idea in the field.

6.1 Limitations

The extended PLEIADES architecture has one major limitation that is worth discussing: while integrating new textual languages or solvers will not, in principle, require any architectural changes, they do imply a non-trivial implementation effort. This is due to the simple fact that a custom CLIF code generator must be fit to a (new or existing) parser for said language, and, for the solver, a code generator from a generic CSP representation has to be designed.

In addition, it remains unclear how VMLs that model constructs such as the assets implementing the concrete leaf features of an SPL or even temporal and state constraints, such as the full Clafer [39] language, can be reasoned upon within the extended PLEIADES architecture. Its current declarative representation of generic CSP and reasoning tasks will likely require further extensions.

6.2 Future Work

We intend to continue this work in several directions. First, we would like to add the ability to generate Generic Constraint Programs at different "levels", i.e., as integer arithmetic problems or as purely boolean problems as a function of the expressivity of the input language. Indeed, UVL models can be annotated with the "theories" (named "levels" in their presentation [64]) necessary for solving them.

Next, we would like to add further fine grained control to the solving strategies employed by the solvers, such as incremental [27] solving and warm-starts (reusing previous complete or incomplete solutions as starting point for search) [28] to support more efficient solving. We suspect it would be beneficial to enable portfolio [3] solving strategies leveraging all solver back-ends simultaneously.

Another avenue we wish to pursue is the treatment of the architecture itself as a software product line, whereby fully configurable distributions could be produced for particular needs, for instance, including a commercial solver if the licenses are available.

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. *Science of Computer Programming* 78, 6 (June 2013), 657–681. https://doi.org/10.1016/j.scico.2012.12.004

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (second ed.). Addison Wesley.

[3] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. 2015. SUNNY-CP: a sequential CP portfolio solver. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. 1861–1867.

[4] Michał Antkiewicz, Kacper Bąk, Alexandr Murashkin, Rafael Olaechea, Jia Hui (Jimmy) Liang, and Krzysztof Czarnecki. 2013. Clafer Tools for Product Line Engineering. In *Proceedings of the 17th International Software Product Line Conference Co-Located Workshops*. ACM, Tokyo Japan, 130–135. https://doi.org/10.1145/2499777.2499779

[5] Ioannis Arapakis, Souneil Park, and Martin Pielot. 2021. Impact of response latency on user behaviour in mobile web search. In *Proceedings of the 2021 Conference on Human Information Interaction and Retrieval*. 279–283.

[6] Franz Baader, Ian Horrocks, Carsten Lutz, and Uli Sattler. 2017. *Introduction to description logic*. Cambridge University Press.

[7] Önder Babur, Loek Cleophas, and Mark Van Den Brand. 2018. Model analytics for feature models: case studies for SPLOT repository. In *2018 MODELS Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS, MODELS-WS 2018*. CEUR-WS. org, 787–792.

[8] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, Vol. 13. 14.

[9] Clark Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer International Publishing, Cham, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11

[10] Maurice H ter Beek, Klaus Schmid, and Holger Eichelberger. 2019. Textual variability modeling languages: an overview and considerations. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B*. 151–157.

[11] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (Sept. 2010), 615–636. https://doi.org/10.1016/j.is.2010.01.001

[12] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2006. A First Step towards a Framework for the Automated Analysis of Feature Models. *Proc. Managing Variability for Software Product Lines: Working With Variability Mechanisms* (2006), 39–47.

[13] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. In *Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005. Proceedings 17*. Springer, 491–503.

[14] Danilo Beuche. 2011. Modeling and Building Software Product Lines with Pure::Variants. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*. ACM, Munich Germany, 1–1. https://doi.org/10.1145/2019136.2019190

[15] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. 1976. *Graph Theory with Applications*. Vol. 290. Macmillan London.

[16] Goetz Botterweck and Andreas Pleuss. 2014. Evolution of Software Product Lines. In *Evolving Software Systems*, Tom Mens, Alexander Serebrenik, and Anthony Cleve (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 265–295. https://doi.org/10.1007/978-3-642-45398-4_9

[17] Barrett R. Bryant, Jeff Gray, and Marjan Mernik. 2010. Domain-Specific Software Engineering. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. ACM, Santa Fe New Mexico USA, 65–68. https://doi.org/10.1145/1882362.1882376

[18] Camilo Correa, Raul Mazo, Andres O. Lopez, and Jacques Robin. 2023. A Lightweight Method to Define Solver-Agnostic Semantics of Domain Specific Languages for Software Product Line Variability Models. In *SOFTENG 2023 - The 9th International Conference on Advances and Trends in Software Engineering*. IARIA: International Academy, Research and Industry Association, Venise, Italy.

[19] Camilo Correa Restrepo. 2024. *ccr185/semantic_translator: Updated Paper Version with Installation Instructions V3*. https://doi.org/10.5281/zenodo.13013170

[20] Camilo Correa Restrepo, Jacques Robin, and Raul Mazo. 2023. Generating Constraint Programs for Variability Model Reasoning: A DSL and Solver-Agnostic Approach. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 138–152.

[21] Douglas Crockford. 2006. *The Application/Json Media Type for JavaScript Object Notation (JSON)*. Request for Comments RFC 4627. Internet Engineering Task Force. https://doi.org/10.17487/RFC4627

[22] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing cardinality-based feature models and their specialization. *Software process: Improvement and practice* 10, 1 (2005), 7–29.

[23] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[24] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77.

[25] Rina Dechter and David Cohen. 2003. *Constraint Processing*. Morgan Kaufmann.

[26] A. O. Elfaki. 2013. Automated Verification of Variability Model Using First-Order Logic. In *Managing Requirements Knowledge*, Walid Maalej and Anil Kumar Thurimella (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 265–289. https://doi.org/10.1007/978-3-642-34419-0_12

[27] Bjorn N Freeman-Benson, John Maloney, and Alan Borning. 1990. An incremental constraint solver. *Commun. ACM* 33, 1 (1990), 54–63.

[28] Robert M Freund. 1991. A potential-function reduction algorithm for solving a linear program directly from an infeasible "warm start". *Mathematical Programming* 52, 1 (1991), 441–466.

[29] Thom Frühwirth and Slim Abdennadher. 2003. *Essentials of constraint programming*. Springer Science & Business Media.

[30] José A Galindo and David Benavides. 2019. Towards a new repository for feature model exchange. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B*. 170–173.

[31] 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 *Proceedings of the 24th Acm International Systems and Software Product Line Conference-Volume b*. 52–55.

[32] José A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated Analysis of Feature Models: Quo Vadis? *Computing* 101, 5 (May 2019), 387–433. https://doi.org/10.1007/s00607-018-0646-1

[33] Dennis F Galletta, Raymond Henry, Scott McCoy, and Peter Polak. 2004. Web site delays: How tolerant are users? *Journal of the Association for Information Systems* 5, 1 (2004), 1–28.

[34] Carla P Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. 2008. Satisfiability Solvers. *Foundations of Artificial Intelligence* 3 (2008), 89–134.

[35] Øystein Haugen, Andrzej Wąsowski, and Krzysztof Czarnecki. 2012. CVL: Common Variability Language. In *Proceedings of the 16th International Software Product Line Conference-Volume 2*. 266–267.

[36] José Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2023. Empirical Analysis of the Tool Support for Software Product Lines. *Software and Systems Modeling* 22, 1 (Feb. 2023), 377–414. https://doi.org/10.1007/s10270-022-01011-2

[37] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. 2018. PySAT: A Python Toolkit for Prototyping with SAT Oracles. In *SAT*. 428–437. https://doi.org/10.1007/978-3-319-94144-8_26

[38] International Organization for Standardization. 2018. *Information Technology – Common Logic (CL) – A Framework for a Family of Logic-Based Languages – ISO/IEC 24707:2018*. Technical Report. International Organization for Standardization, Geneva, CH. 70 pages.

[39] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2018. Clafer: Lightweight Modeling of Structure, Behaviour, and Variability. *The Art, Science, and Engineering of Programming* 3, 1 (July 2018), 2. https://doi.org/10.22152/programming-journal.org/2019/3/2

[40] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study:*. Technical Report. Defense Technical Information Center, Fort Belvoir, VA. https://doi.org/10.21236/ADA235785

[41] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is there a mismatch between real-world feature models and product-line research?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 291–302.

[42] Tim Korson and John D McGregor. 1990. Understanding object-oriented: A unifying paradigm. *Commun. ACM* 33, 9 (1990), 40–60.

[43] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. 2018. KernelHaven – An Experimentation Workbench for Analyzing Software Product Lines. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 73–76. https://doi.org/10.1145/3183440.3183480 arXiv:2110.05858 [cs]

[44] Charles Krueger and Paul Clements. 2018. Feature-Based Systems and Software Product Line Engineering with Gears from BigLever. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 2*. 1–4.

[45] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. 2015. Sat-based analysis of large real-world feature models is easy. In *Proceedings of the 19th International Conference on Software Product Line*. 91–100.

[46] John W. Lloyd and John Wylie Lloyd. 1993. *Foundations of Logic Programming* (2., extended ed., 1. corr. print ed.). Springer, Berlin Heidelberg.

[47] Jacob Loth, Chico Sundermann, Tobias Schrull, Thilo Brugger, Felix Rieg, and Thomas Thüm. 2023. UVLS: A Language Server Protocol For UVL. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference-Volume B*. 43–46.

[48] Mike Mannion. 2002. Using First-Order Logic for Product Line Model Validation. In *Software Product Lines: Second International Conference, SPLC 2 San Diego, CA, USA, August 19–22, 2002 Proceedings*. Springer, 176–187.

[49] Robert C Martin. 1996. The dependency inversion principle. *C++ Report* 8, 6 (1996), 61–66.

[50] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. ACM, Orlando Florida USA, 761–762. https://doi.org/10.1145/1639950.1640002

[51] Raphael Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. 2011. A Formal Semantics for Feature Cardinalities in Feature Diagrams. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*. ACM, Namur Belgium, 82–89. https://doi.org/10.1145/1944892.1944902

[52] Fiona Fui-Hoon Nah. 2004. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology* 23, 3 (2004), 153–163.

[53] Damir Nešić, Jacob Krüger, Ștefan Stănciulescu, and Thorsten Berger. 2019. Principles of feature modeling. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 62–73.

[54] Mahdi Noorian, Alireza Ensan, Ebrahim Bagheri, Harold Boley, and Yevgen Biletskiy. 2011. Feature Model Debugging Based on Description Logic Reasoning.. In *Proceedings of the 17th International Conference on Distributed Multimedia Systems, DMS 2011, October 18-20, 2011, Convitto della Calza, Florence, Italy*, Vol. 11. Citeseer, 158–164.

[55] Object Management Group. 2014. *Model Driven Architecture (MDA) – MDA Guide rev. 2.0*. Technical Report. Object Management Group.

[56] Object Management Group. 2016. *OMG Meta Object Facility (MOF) Core Specification*. Technical Report. Object Management Group.

[57] Object Management Group. 2021. *Semantics of a Foundational Subset for Executable UML Models (fUML)*. Technical Report. Object Management Group.

[58] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques* (1st ed ed.). Springer, New York, NY.

[59] Leonard Richardson and Sam Ruby. 2008. *RESTful web services*. " O'Reilly Media, Inc.".

[60] Anna Schmitt, Christian Bettinger, and Georg Rock. 2018. Glencoe–a Tool for Specification, Visualization and Formal Analysis of Product Lines. In *Transdisciplinary Engineering Methods for Social Innovation of Industry 4.0*. IOS Press, 665–673.

[61] Michael Schulze and Robert Hellebrand. 2015. Variability Exchange Language-A Generic Exchange Format for Variability Data.. In *Software Engineering (Workshops)*. 71–80.

[62] Ramy Shahin, Robert Hackman, Rafael Toledo, S Ramesh, Joanne M Atlee, and Marsha Chechik. 2021. Applying declarative analysis to software product line models: an industrial study. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 145–155.

[63] Steve Cook, Conrad Bock, Pete Rivett, Tom Rutt, Ed Seidewitz, Bran Selic, and Doug Tolbert. 2017. *Unified Modeling Language (UML), Version 2.5.1*. Technical Report. Object Management Group.

[64] 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 *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*. ACM, Leicester United Kingdom, 136–147. https://doi.org/10.1145/3461001.3471145

[65] Chico Sundermann, Stefan Vill, Thomas Thüm, Kevin Feichtinger, Prankur Agarwal, Rick Rabiser, José A Galindo, and David Benavides. 2023. UVLParser: Extending UVL with Language Levels and Conversion Strategies. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference-Volume B*. 39–42.

[66] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 79 (Jan. 2014), 70–85. https://doi.org/10.1016/j.scico.2012.06.002

[67] University of Seville, University of Malaga, and University of Ulm. 2023. UVLHub. https://www.uvlhub.io/. Accessed: 2024-03-27.

[68] VariaMos Team. 2023. VariaMos Framework. https://variamos.com/. Accessed: 2023-03-27.

[69] Angela Villota. 2022. *Coffee : A Framework Supporting Expressive Variability Modeling and Flexible Automated Analysis*. Ph. D. Dissertation. Université Panthéon-Sorbonne - Paris I.

[70] Markus Voelter and Eelco Visser. 2011. Product Line Engineering Using Domain-Specific Languages. In *2011 15th International Software Product Line Conference*. IEEE, Munich, Germany, 70–79. https://doi.org/10.1109/SPLC.2011.25

[71] Eric Zinda. [n. d.]. *Python and Other Programming Languge Integration for SWI Prolog*. https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/mqi.html%27)

Received 28 March 2024; revised 31 July 2024; accepted 31 July 2024