Generating Constraint Programs for Variability Model Reasoning: A DSL and Solver Agnostic Approach

Camilo Correa, Jacques Robin, Raùl MAZO

▶ To cite this version:

Camilo Correa, Jacques Robin, Raùl MAZO. Generating Constraint Programs for Variability Model Reasoning: A DSL and Solver Agnostic Approach. ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, Oct 2023, Lisboa, Portugal. 10.1145/3624007.3624060 . hal-04330769

HAL Id: hal-04330769
https://hal.science/hal-04330769v1

Submitted on 8 Dec 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Generating Constraint Programs for Variability Model Reasoning: A DSL and Solver Agnostic Approach

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

Jacques Robin
jacques.robin@esiea.fr
ESIEA
Paris, France

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

Abstract

Verifying and configuring large *Software Product Lines (SPL)* requires automation tools. Current state-of-the-art approaches involve translating *Variability Models (VM)* into a formalism accepted as input by a constraint solver. There are currently no standards for the *Variability Modeling Languages (VML)*. There is also a variety of constraint solver input languages. This has resulted in a multiplication of ad-hoc architectures and tools specialized for a single pair of VML and solver, fragmenting the SPL community. To overcome this limitation, we propose a novel architecture based on model-driven code generation, where the syntax and semantics of VMLs can be declaratively specified as data, and a standard, human-readable, formal pivot language is used between the VML and the solver input language. This architecture is the first to be fully generic by being agnostic to both VML and the solver paradigm. To validate the genericity of the approach, we have implemented a prototype tool together with declarative specifications for the syntax and semantics of two different VMLs and two different solver Families. One VML is for classic, static SPL (Feature Model) and the other is for run-time reconfigurable *dynamic* SPL with soft constraints to be optimized during configuration. The two solver families are *Constraint Satisfaction Program (CSP)* and *Constraint Logic Programming (CLP)*.

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

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GPCE, October 22–27, 2023, Cascais, Portugal
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
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 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software Product Lines Engineering (SPLE) [38] is a collection of engineering techniques for managing the design, development, and subsequent evolution of large sets of software products that share partially overlapping sets of requirements and reusable software assets implementing them. Considered as one whole, these products form the eponymous ***Software Product Line (SPL)***. Crucially, each product in a SPL is a *variant* that includes a specific subset of the requirements and reusable assets. This variability makes SPLE unique: it needs to be managed across the entire product line to control the quality and consistency of the products. To carry this out, SPLE prescribes the construction, verification and maintenance of an additional software artifact called a ***Variability Model***, that represents the relationships holding among the SPL's variable requirements and the reusable assets implementing them. These models have been used for three main distinct purposes:

- ***Software Mass Customization*** [29] from a baseline software version with automated variant code generation from reusable artifacts.
- ***Design Space Exploration*** [32], *i.e.,* determining which set of design choices best satisfices[1] a set of hard (*i.e.*, must have) and soft (*i.e.*, nice to have) requirements.
- Context-aware ***Autonomic Adaptation*** [31, 50] through runtime self-reconfiguration.

The third purpose involves embarking the VM as a runtime artifact to support re-exploring the design space during operations to find an alternative configuration that restores requirement satisficing following an operational context change that rendered the current configuration inadequate. SPLs for such self-adaptive systems are called ***Dynamic SPLs (DSPL)*** [25].

[1] *Satisfice* is a portmanteau word derived from *satisf*ying and suf*fice*, conveying satisfaction to a sufficent extent rather than necessarily completely.

In all three cases, the models support the semi-automatic derivation of correct *configurations, i.e.,* choices of cohesive and consistent requirements that satisfy the business, technical and regulatory constraints captured by the Variability Model. Real-life, industrial SPLs are too large and complex to be *manually* verified, troubleshooted and correctly configured [33]. To make things worse, these tasks need to be repeated after each evolution of the SPL during its life-cycle that routinely spans over multiple decades [11]. Automated reasoning must thus be used to repeatedly verify, troubleshoot and (re)configure an SPL at initial design-time, evolution-time and even runtime throughout its life-cycle. In most cases, the *augmented* intelligence [20] flavor of automated reasoning is required, in which the reasoning must be explainable to the team of human engineers and operators managing the SPL and that have the final say concerning each configuration or defect correction choice.

Today, there is no accepted standard for SPL Variability Models, so every SPLE automation tool uses its own ***Domain-Specific Variability Modeling Language (DSVML)***. However, despite superficial syntactic differences, at the semantic level, most DSVMLs used for Software Mass Customization define cohesive requirement sets, generally called *features*, and share four key expressive capabilities:

SFM1 Organize these features into abstraction and composition hierarchies and associate the lowest level ones with reusable and composable concrete software assets implementing them.
SFM2 Distinguish between *mandatory* features shared by all configurations from *optional elements* specific to strict configuration space subsets.
SFM3 Specify ranges of alternative possibilities for the refinement of a higher-level feature into a set of lower-level features.
SFM4 Specify simple required co-occurence or exclusion constraints between two features across the abstraction hierarchy.

DSVML providing such expressive capabilities are generally referred to as ***Simple Feature Models (SFM)***. Four additional expressive capabilities are provided by DSVML which are often called ***Extended Feature Models (EFM)***:

EFM1 Structure features into sets of attributes of various types rather than limiting them to Boolean variables.
EFM2 Specify ranges of alternative possible values for those attributes.
EFM3 Specify multiplicity constraints on those attributes and on relationships among features.
EFM4 Specify complex constraints on the values of attributes of features located anywhere in the abstraction hierarchy.

Whether SFM or EFM, all models used for Software Mass Customization only contain *hard* constraints that must be collectively consistent and fully satisfied by all valid configurations. In contrast, models used for Design Space Exploration and Autonomic Adaptation also contain *soft* constraints to satisfice as much as possible rather than necessarily fully; thus their modeling languages need to be semantically more expressive.

A wide variety of approaches have been proposed to implement SPL model verification and model-guided SPL configuration in SPLE automation tools. Just like for Variability Modeling Languages, there is also currently no accepted standard API for such tools. Nevertheless, the overwhelming majority of them include a translator of the DSVML to some logical ***Knowledge Representation Language (KRL)***. This allows them to reuse practically scalable ***Inference Engines (IE)*** developed over the last 50 years for formal software engineering and artificial intelligence. Four main paradigms of such IEs have been used to automate SPL model verification and model-guided configurations:

- ***SATisfiability (SAT)*** solvers [24] and their extensions with ***Satisfiability Modulo Theories (SMT)*** [16].
- ***Constraint Satisfaction Problem (CSP)*** solvers and their extensions for ***Constraint Optimization Problems (COP)*** [17].
- ***Logic Programming (LP)*** engines and their ***Constraint LP (CLP)*** extensions [21].
- ***Description Logic (DL)*** engines and their semantic web ontology reasoning extensions [5].

Notably, the foundational feature-based SPLE tool FODA used an LP engine [27], the original version of VariaMos [47] used a CLP engine, FeatureIDE [43], FlamaPy [22], Splot [35], Glencoe [42], Kernel Haven [28] and pure::variants [10], all use a SAT solver, Familiar can use either an SMT or a CSP solver [2], COFFEE used a CSP solver [48] and AUFM used a DL engine [37].

Each pair of KRL and IE from these paradigms corresponds to a different trade-off in terms of semantic expressiveness, inference scalability and reasoning explainability. The best KRL for a given SPL reasoning task is thus very much dependent on both the nature of the task and the semantic expressiveness of the DSVML used to model variability [7]. Since SPLE is a heavy upfront investment method whose "Return on Investment" takes a fairly long time before becoming tangible [38], SPLE projects have long life cycles. Therefore, both the expressiveness requirements of a DSVML and the automated reasoning tasks to analyze the VM and correctly (re)configure the SPL can evolve significantly during its life cycle.

The main common limitation of the state-of-the-art SPLE automation tools listed above is their *ad-hoc* architectures that tightly couples a single DSVML with an IE from a given automated reasoning paradigm. This impedes one from making the choice of IE follow in lockstep the evolution of the DSVML and reasoning task requirements at a cost that is low

enough to avoid denting the long-term benefits of adopting SPLE.

In an attempt to overcome this severe limitation of current state-of-the-art SPLE tools, we try, in this paper, to answer the following open research question:

How to architect an SPLE automation tool so that IEs from various paradigms can be seamlessly plugged in and out of it at minimal development cost to adapt the tool's reasoning capabilities to the evolution, throughout the SPL's life cycle, of both the semantic expessiveness of the DSVMLs that it supports for variability modeling and the analysis and configuration automation tasks to run on the Variability Model?

Our first research hypothesis to answer this question is that such an architecture must satisfy the following requirements:

- **REQ1** Support low-cost extension of existing DSVMLs and addition of new DSVMLs.
- **REQ2** Support low-cost addition of new automated reasoning tasks to run on the VM.
- **REQ3** The architecture must be agnostic *w.r.t.* the logical KRL and IE paradigm used for automated reasoning on the VM, supporting low-cost addition of interoperability with solvers from different paradigms.
- **REQ4** The architecture must be agnostic *w.r.t.* the DSVML editor tool, accepting the VM as input data exportable from multiple popular editors.

To satisfy these requirements, we propose the following design principles, inspired from ***Model-Driven Engineering*** [41]:

- **DP1** The concrete and abstract syntaxes of the DSVML should be decoupled from one another.
- **DP2** They both should be *declaratively* specified as data in a widely used exchange format, rather than hard-coded in the SPLE tool.
- **DP3** The semantics of the DSVML should also be *declaratively* specified as data in a widely used exchange format encoding a mapping from the abstract syntax elements to expressions in a formal KRL.
- **DP4** The set of reasoning automation tasks to be run on the VM should also be *declaratively* specified as data in a widely used exchange format, which must furthermore be decoupled from any specific IE KRL.
- **DP5** The many-to-many translation from the multiple DSVMLs to the multiple IE KRLs should be decoupled into a pipeline of N many-to-one transformations to a *standard pivot intermediate language* followed by M one-to-many transformations from this pivot, to avoid the combinatorially explosive cost of developing and evolving NxM direct DSVML to IE KRL transformations.
- **DP6** This standard pivot language must be easily interpretable by a wide range of stakeholders.

Our second research hypothesis is that REQ1 and REQ4 can be satisfied by the combination of DP1 to DP3, REQ2 can be satisfied by DP4 and REQ3 by that of DP5. In the rest of the paper, we attempt to verify our two hypotheses.

To do so we proceed as follows. In section 2, we start by presenting two SPLE reasoning task examples that we have used as a first step toward validating these hypotheses. While they are small enough to fit in this article, they are purposely representative of very different VM language families used for very different VM purposes. In that section, we also explain how these tasks can be carried out by leveraging, as intermediate pivot language between the two different input DSVML and any logical IE input KRL, the ISO standard for logical IE interoperability ***CLIF (Common Logic Interchange Format)*** [1] following the original proposal of [14]. Next, in section 3, we propose a detailed SPLE tool architecture following the design principles DP1 to DP5 listed above. We then discuss the prototype SPLE tool ***PLEIADES (Product Line Engineering Intelligent Assistant for Defect detection Explanation and Solving)*** that we (a) implemented to show the practical feasibility of this architecture and (b) tested on the two example tasks presented in section 2. In section 5, we then compare PLEIADES with state-of-the-art SPLE tools. Since none of them aimed to satisfy requirements *REQ1* through *REQ4*, nor explicitly followed design principles DP1 to DP5, this comparison is grounded on various tool versatility criteria which are met by following these principles. In section 6 we discuss the limitations of both the presented architecture and its current implementation in terms of satisfaction of requirements REQ1 to REQ4 and the future work that we intend to carry out to overcome them. Finally in section 7, we conclude by recapitulating the original contributions of the presented research.

2 Background and Running Examples

To illustrate our approach on concrete examples, we now present two of them. The first concerns a defect detection verification task on an SFM. The second concerns an optimal reconfiguration search for a DSPL that leverages a DSVML specifically designed for that purpose and presented in [40]. They thus represent two intentionally distant points in the space of the VM diversity encountered in the literature.

2.1 Variability Modeling Languages

2.1.1 Simple Feature Models. In Fig. 1, we present a minimalist SFM example. Its graphical concrete syntax shows features as rectangular vertices in a directed graph and the hierarchical decomposition of features in mandatory and optional sub-features as edges ending with a filled or empty circle (respectively). This decomposition forms a tree. Edges ending with an arrow represent exclusion or co-occurence

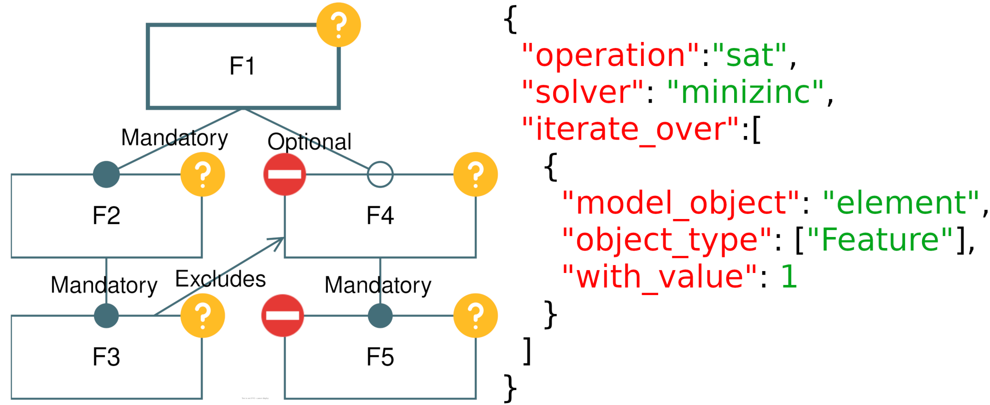

Figure 1. First part of our running example with a synthetic Feature Model. The query specifying the iterative search for dead features in the model is beside the model.

constraints among feature pairs in different branches of the tree.

Verification reasoning tasks on such VMs generally search for defects, which were comprehensively categorized in [7]. In figure 1 we show a SFM which contains two dead feature defects. A feature is said to be "dead" if, although it is present in the VM, it can never be selected in any valid configuration due to contextual constraints relating to other features. Following principle DP4, the text beside the SFM diagram is the declarative specification in JSON [15] of the search for dead feature defects reasoning task. The one-way widgets on the top-left features F4 and F5 visualize the result of this task. It detects that F4 and F5 are dead. F4 is dead as excluded by the selection of F3, which must be selected in all configurations as a mandatory descendant of the top-level feature F1. F5 is also dead as a child of F4, a dead feature.

2.1.2 Sawyer et al.'s DSPL VML.

In Fig.2, we present a second example of reasoning task. This time it is a search for an optimal reconfiguration leveraging the context-aware DPSL VM of a flood early-warning system. The vocabulary of concepts and relations of this DSPL DSVML [40] is much more extensive than that of SFMs used in the first example. They are the following:

- *Hard goals*, shown as green parallelograms, determine the functional requirements of the system and are analogous to features in feature models. Hard goals are structured in a decomposition hierarchy where higher level ones can be achieved by achieving all their lower level components.
- *Soft Goals*, shown as blue clouds, encode the non-functional requirements of the system and can be satisficed on a 0 to 4 scale, which is encoded as "--", "-", "=", "+", "++" in the model. They are themselves structured into a decomposition hierarchy. The level of satisficing of a higher level soft goal in this hierarchy is the average of the satisficing level of its lower level components.
- *Context Variables*, shown in blue rectangles on the right, encode the state of the system's context among symbolic value enumerations.
- *Operationalizations*, shown as gray hexagons, are concrete software assets that can implement the hard goals.
- *Bundles*, shown as white circles, contain integer range expressions for the multiplicity of the operationalizations that can implement a hard goal.
- *Claims*, shown as white trapezoids, express the level to which operationalizations satisfy soft Goals as a function of which has been selected.
- *Soft Influences*, shown as grey ellipses, relate the context variables to the soft goals, and determine the required level of satisfaction when the given state is determined by the context, e.g., if CV1 is "Low", the required level of satisfaction of SG5 is "++".

The VM in Fig. 2 models the various redundant means of communications available to transfer sensor data in a distributed autonomic flood early-warning system. It also supports determining which set of means represent the best trade-off between energy consumption, fault tolerance and prediction accuracy in various flood risk contexts.

The contrast between the VM of figures 1 and 2 highlights the great diversity of DSVMLs and the reasoning tasks to carry out on them.

2.2 CLIF and Model Transformation

The language used to encode the semantics of the different variability modeling languages, CLIF, is part of a larger family of (logic) languages "designed for use in the representation and interchange of information and data among [...] computer systems" [1] named *Common Logic*. CLIF has not been created *ex nihilo*, but rather, is a simplified descendant of the Knowledge Interchange Format Version 3 [23] KRL. It

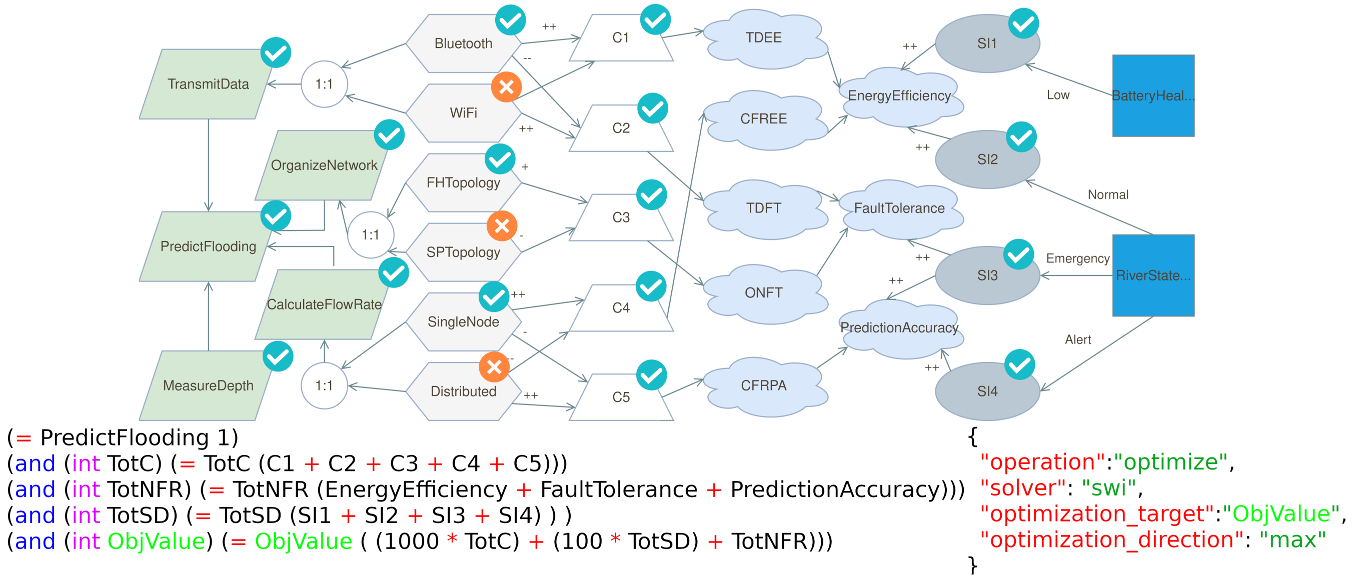

Figure 2. Second part of our running example with a model using Sawyer et al.'s DSPL VML. This model reproduces the original example from the paper [40] for a flood early-warning system. As with our other example, in addition to a reasoning query to solve (on the bottom right), this time, an optimization problem to find an optimal configuration whose additional constraints are shown below the model (bottom left).

uses a LISP [34] derived syntax to represent logical expressions. For example $(a \wedge b)$ would be rendered as `(and a b)` where a and b can also be nested subexpressions.

Figure 3 exemplifies the overall mechanism underlying the architecture. Given a fragment of a diagram and the corresponding semantics specification given as a JSON [15] object, our architecture would allow then for the construction of the human- and machine-readable semantics in CLIF. It would then allow one to target different solvers based purely on the CLIF semantics. The object, then, of this article, is to provide a detailed account of how this is achieved and how this architecture overcomes the key limitation of ad-hoc architectures and tools specialized for a single pair of variability modeling language and solver.

2.3 Reasoning Tasks

One of the primary goals of our approach is to allow practitioners to perform reasoning tasks on their models through a simple specification mechanism to avoid the need to understand the particular details of the underlying implementation or how to perform meta-programming, for example. Both of our running examples (Figs. 1 and 2) include a JSON object that encodes the reasoning request that is to be made about the model. For the Feature Model, we perform an iterative reasoning task that corresponds to searching for dead features [7]. This object informs the backend of the operations required for the reasoning task. The first attribute **"operation"** determines what type of problem will be solved(e.g. a satisfiability check, finding a concrete solution, or solving an optimization problem). The second, **"solver"**, tells the system which runtime to utilize for the query.

The two attributes above are always required and set the main parameters of the system. The (optional) attribute **"iterate_over"** allows the user to perform a succession of queries on the model. Its value is a list of objects that *determine* how the elements will be modified in each run of the iteration. In the concrete case of Fig. 1, it will iterate over all *"Features"*, setting their associated variable to 1, i.e., selected. The result of the reasoning task is then overlayed on the model as a red prohibition/one way sign showing that those two features are not selectable in any valid configuration.

The other attributes depicted in Fig. 2, **"optimization_target"** and **"optimization_direction"** serve a different purpose. They parametrize the search for optimal solutions where the optimization expression is a variable in the desired direction, e.g. maximizing it. At the very bottom of Fig. 2 are the textual constraints that complement the graphical model and permit the construction of the optimization problem. They are written in CLIF right alongside the Graphical models and are part of them, exhibiting our support for hybrid textual/graphical models. The key takeaway of these constraints is that they define a set of equations related to the model elements and ultimately allow one to equate the optimization variable to a complex expression. In this case, our objective variable is **ObjValue** which is defined as a weighted (shown below) sum over the satisfaction level of claims, soft influences and soft goals as defined above. Since we seek to simulate the behaviour of a context-aware Dynamic SPL, we also provide

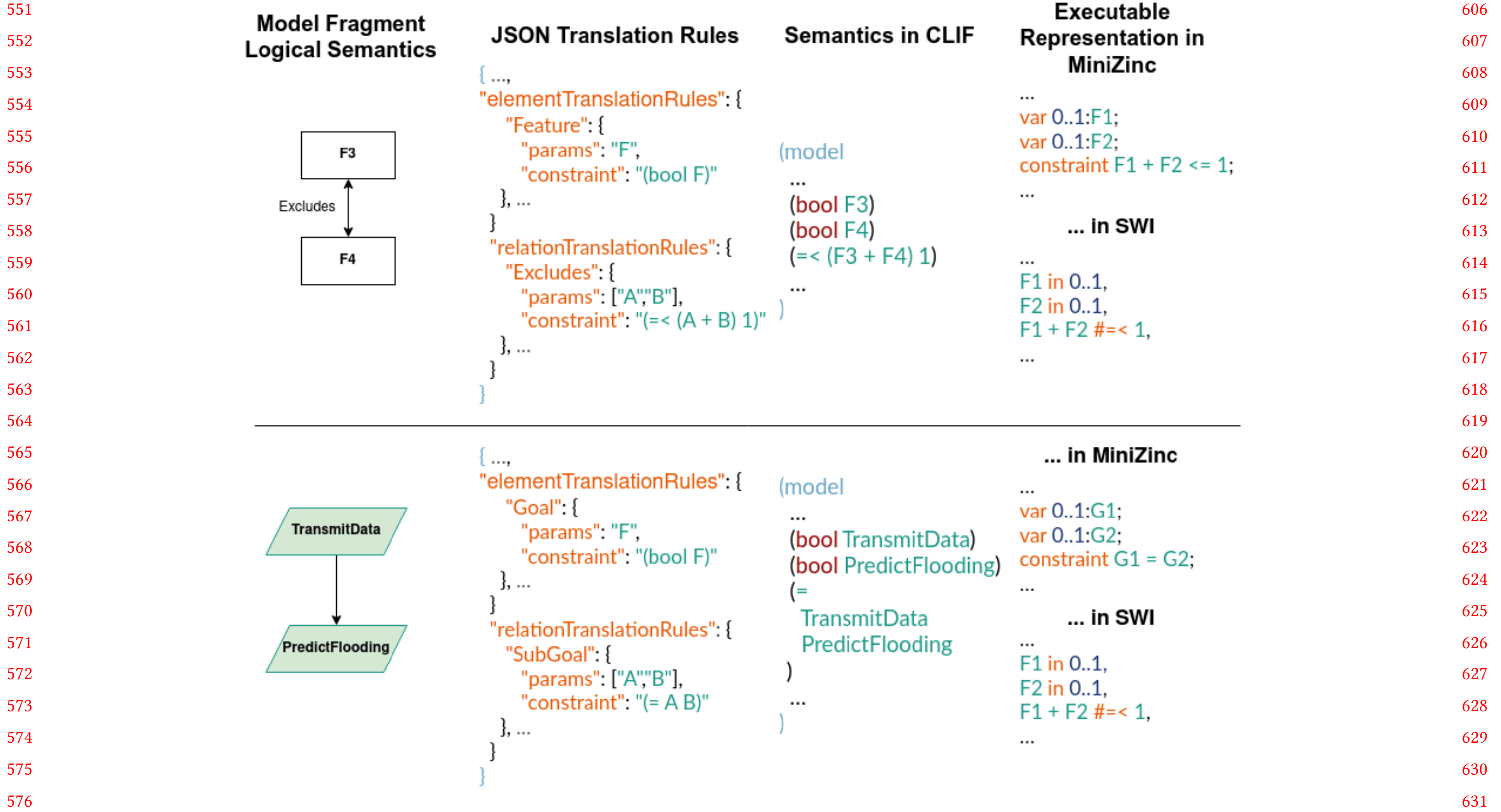

Figure 3. Concrete examples of translation using the translation rules specification for Feature Models (top) and Sawyer et al.'s DSPL Variability Modeling Language (bottom).

the context values to influence the search for an optimal solution by setting the context variables (blue boxes on the right) to one of their possible values.

```
(and (int ObjValue)
    (= ObjValue ((1000*TotC)+(100*TotSD)+TotNFR)))
```

3 Overview of the Architecture

As mentioned in the introduction, the key contribution of this article is a generic architecture for reasoning on VMs that is agnostic to both the input VM and the IE used for reasoning (and hence its KRL). We propose a client-server web architecture in which variability model verification and configuration reasoning functionalities are provided as web services accessible through a REST API endpoint. With this approach, multiple VM editor clients can send HTTP requests for VM verification or SPL configuration tasks and receive as response the result of the automated reasoning performed by the constraint solvers hosted by the server. This choice of a web service architecture allows fully separating the concern of editing a VM from the concern of reasoning on it. It also insures full decoupling of the implementation platforms respectively used for (a) VM editing, (b) translation of VM reasoning requests into constraint solver inputs and back from constraint solver outputs into VM reasoning responses and (c) constraint solving. Additionally, it provides an installation-free friction-less usage of the reasoning services.

3.1 High-Level Component Structure and Control Flow of the Architecture

The UML diagram in Fig. 4 shows the structural model of the PLEIADES architecture that we propose for SPLE reasoning services. It shows its main components, the signatures of the operations that they implement, together with the data types of each signature parameter. While this diagram is technically a class diagram, to remain as agnostic as possible with respect to the modeling or programming paradigm, it contains the very general concepts of components (*a.k.a.*, service, module or package in different implementation platforms) and data types, rather than classes, which may have implied the adoption of class-based object-orientation to implement our architecture.

The `SPLE Reasoning Web Service` is the top-level component of the architecture. It answers web requests that contain as their payload a serialized JSON representation of the reasoning request to execute on the server. As shown at the top-left of Fig. 4, this `Serialized Reasoning Request` includes four top-level properties: (a) the `VM` to analyze, (b)

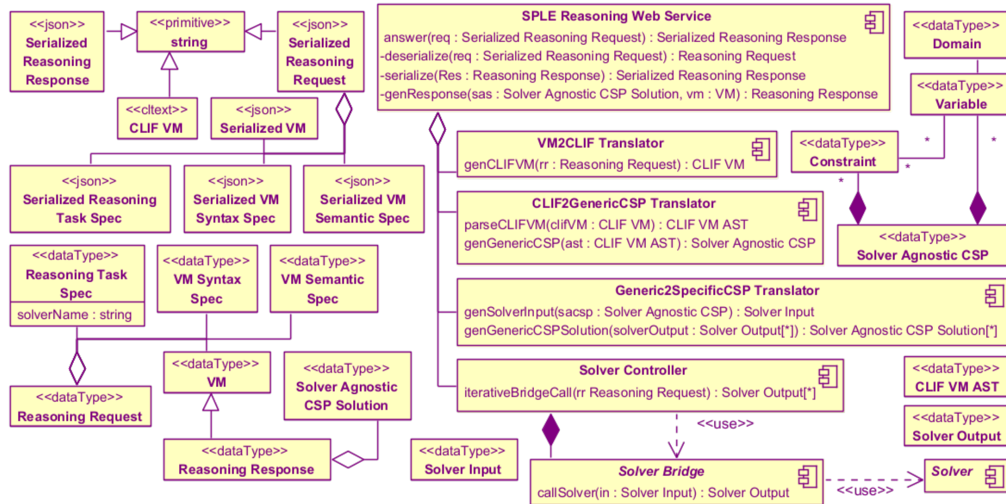

Figure 4. High-level components and data types of the PLEIADES architecture

the specification of the reasoning task to carry out for the analysis, (c) the abstract syntax specification of the VML used by the VM to model the variability of an SPL, and (d) the semantic specification of the VML in the form of a mapping between VML model elements and logical sentences in CLIF. It is the fact that the reasoning request comes with declarative specifications of the abstract syntax and formal semantics of the VML in which the VM is modeled that allows VM reasoning services following this PLEIADES architecture to be VML agnostic. And it is also the fact that the VML's formal semantics are expressed in CLIF, a standard for logical inference engine interoperability, whose expressiveness subsumes that of all constraint solvers widely used for SPLE automated reasoning, that allows VM reasoning services following the PLEIADES architecture to be solver agnostic.

In addition to its public `answer` operation endpoint, the `SPLE Reasoning Web Service` also encapsulates three private operations: one to `deserialize` the reasoning request JSON string into an instance of the `Reasoning Request` data type, one to generate the instance of the `Reasoning Response` data type from the result returned by the solver called to answer the `Reasoning Request` in a solver agnostic format and one to serialize this `Reasoning Response` data type instance into a JSON string.

This top-level component also has access to the operations of its nested components. Let us now review them in top-to-bottom order as depicted in Fig. 4. The first is the `VM2CLIF Translator` which translates the `VM` data structure contained in the `Reasoning Request` data structure into a CLIF text by simultaneously leveraging the VM syntax and semantic specifications that accompany the VM in the `Reasoning Request`.

The second is the `CLIF2GenericCSP Translator` which translates the CLIF text representing the semantics of the `VM` into a semantically equivalent solver-agnostic CSP representation of the `VM`. As shown at the top right of Fig.4, this representation is simply a set of constraints relating variables, each one associated with its domain of possible values. This translation occurs in two steps. The first is to parse the CLIF text into an ***Abstract Syntactic Tree (AST)*** and the second is to generate, from this AST, the constraints, variables and domain data types of the solver-agnostic CSP.

The third is the `Generic2SpecificCSP Translator` which translates this solver-agnostic representation of a CSP into one accepted as input by the solver chosen in the `Reasoning Task Specification`. This component is also used to translate the solver output back in the other direction into a `Solver Agnostic CSP Solution`. Note that due to its purely declarative, relational and intentional nature, a CSP and its solution can be uniformly represented by the same three data types: `Constraint`, `Variable` and `Domain`. A CSP solution is merely a CSP with less constraints and more variables with domains reduced to a singleton [17]. As shown at the bottom-left of 4, the `Reasoning Response` data type associates the value of those singleton domains with the VM element represented by the CSP variable whose domain has been reduced to a single value. That value is injected in the `Reasoning Response` that is then serialized and sent back to the client VM editor.

The fourth is the `Solver Controller`, the most complex component of the architecture. Understanding its role requires realizing that many VM verification tasks cannot be directly executed through a single call to a constraint solver. They rather require meta-programming an iteration over the VM elements in which, at each step, the initial CSP representation of the VM is modified by adding or removing some

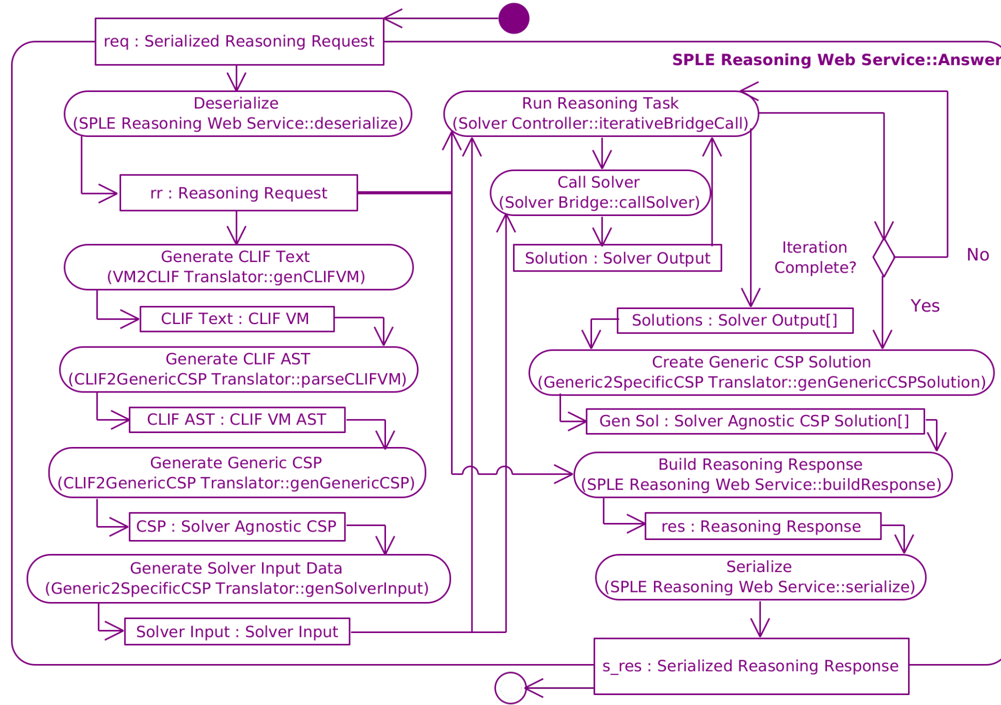

Figure 5. Activity diagram detailing the operation of our achitecture.

constraints and the solver is called on this modified CSP [39]. Depending on the result returned by the solver at each step, the iteration proceeds to the next step or stops. During the iteration, the results of each solver call are accumulated in a set. The `IterativeBridgeCall` operation of the `Solver Controller` implements this iteration. It returns a set of `Solver Output` data type instances.

As an example of this need for iteration, let us consider the search for so-called *dead features* in a feature model: *i.e.,* those that cannot be selected in any valid configuration. The declarative specification of this task as a JSON object is shown beside the model of Fig.3. It defines a strategy for the `Solver Controller` to find dead features in the VM [7]. It consists of iterating over the VM features, and, for each of them, adding to the CSP, representing the VM, the new constraint that this feature is selected (`"with_value": 1`) and checking whether this makes the CSP unsatisfiable. To be able to repeatedly call the requested solver, the `Solver Controller` encapsulates a set of bridges, one for each solver to be integrated in the `SPLE Reasoning Web Service`. The role of each bridge is to start a new instance of the solver, call its API to pass the CSP to solve at each iteration of the `IterativeBridgeCall` operation and pass the result of each such call back to the `Solver Controller`.

The control flow of the top-level `answer` operation of the root `SPLE Reasoning Web Service` component is modeled by the UML activity diagram of Fig.5. It shows the input and output parameter of the `answer` and in what order this operation internally calls the other operations of the architecture. It also shows the data structures that these calls exchange as arguments. At a high-level, this activity is divided in three main phases. The first is the pipeline on the left side of Fig.5 that progressively transform the `Serialized Reasoning Request` received from the VM web client editor into an input for the solver (`Solver Input`) chosen as a property of the `Serialized Reasoning Request`. The second is the loop of calls to the solver made by the `Solver Controller` shown in the top right quadrant of Fig.5. The third and last phase is another pipeline, shown in the bottom right quadrant of Fig.5, that translates the solver outputs accumulated during the iteration into a `Serialized Reasoning Response` to send back to the web client editor.

3.2 Revisiting Requirements and Design Principles

In light of the requirements and design principles we've outlined in the Introduction, it is important to highlight how our architecture reflects these principles and above all meets the requirements. Design principles **DP1** and **DP2** are reflected first in the clear separation of the data types shown in Fig. 4, where every concern has its own dedicated data structure (and hence treatment in the function signatures in the components). They are also reflected in the envisioned operation of the architecture shown in Fig. 5, where they are taken as user-specified specifications (as part of the Reasoning Request) instead of being hard-coded into the architecture. The

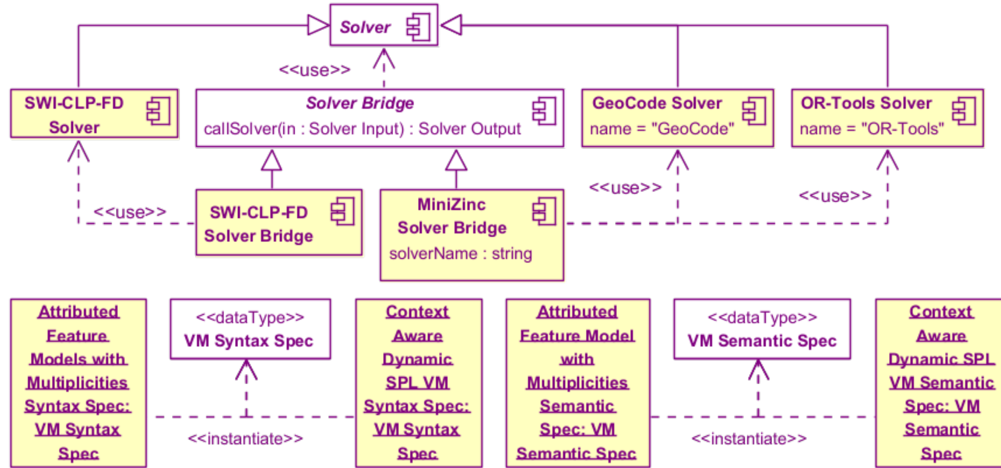

Figure 6. Current PLEIADES prototype instantiating the generic architecture

same is also true for Principle **DP3** when it comes to the specification of the semantics. As for the widely used exchange format, JSON [15] is well supported and certainly widely used. In addition, concerning the enconding into a formal KRL, this is covered by the translation into CLIF based on the semantics specification, allowing virtually any DSVML with first order semantics to be translated into it. Considering that we have met these principles with our architecture, this would cover requirements **REQ1** and **REQ3**.

Design Principle **DP4** is handled in much the same way as **DP2**, wherein JSON is again used to allow the user to tailor his specification for his reasoning task with no need to manually hard-code it in the system, as described in Section 2.3. This would then cover requirement **REQ2**. Principle **DP5** is, firstly, contingent on the earlier principles being met, and, second, is covered by the VM transformation outlined in Fig. 5. Indeed, a significant portion of the architecture is dedicated to enabling this translation, as evidenced by both the component solely dedicated to it (`VM2CLIF Translator` shown in Fig. 4) and the subsequent components that are dedicated to the treatment of the generated CLIF to arrive at the solvers' input. To be clear, what this achieves is to completely decouple integrating DSVMLs and IEs into the architecture by linking both through CLIF and meeting requirement **REQ3**.

Design principle **DP6** stands alone as it is tied to a consideration that is not purely functional, as are the requirements, but rather, it is tied to the very nature of the architecture, that is, easing the use and analysis of VMs by different stakeholders without needing to dive into the details of the specific underlying solvers. This also aids users to construct (and debug) new DSVMLs as they can examine the semantics in CLIF of their models independent of any particular solving technology.

4 Prototype

To validate our architecture and its feasibility for constructing a reasoning platform that covers all of the requirements outlined above, we have implemented a prototype in Python based on this architecture. Fig. 6 shows how the prototype specializes the components outlined in the architecture. This prototype's internal component structure closely reflects the architecture. A key element of our prototype is the integration of multiple solvers from different solver families: SWI Prolog [51] and MiniZinc [36] (which allows us to target multiple Constraint and Integer Programming Solver libraries). To implement the pivot language, we wrote our own "CLIF Parser" with the TextX library [18] based on the grammar described in the latest version of the ISO standard [1].

We demonstrate our approach within an open-source tool called VariaMos [47] for all visualization concerns, and for the specification of the concrete visual syntax, the abstract syntax and the formal semantics in JSON format. The examples shown in Figs. 1 and 2, and the results shown, were all done within this tool. We have modified the tool so that the reasoning requests can be made directly from it by specifying the address of the reasoning web service. We have made the prototype available on GitHub as an open source tool[2].

5 Related Work

5.1 State of the art VM tools

In this section, we examine state-of-the-art tools with aims similar to ours that are currently available, mature and well documented as reported by a recent survey [26], with the addition of some others that we consider particularly relevant. The tools we've identified are: Feature IDE [43], its related project FAMILIAR [2], FlamaPy [22], the COFFEE Framework [48], SPLOT [35], Glencoe [42], ClaferTools [4], Kernel

[2]https://github.com/ccr185/semantic_translator. The use manual is in the repository's wiki page: https://github.com/ccr185/semantic_translator/wiki

Table 1. Characteristics of State of the Art Tools

		Tools										
		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)
Architecture	Standard Modeling Language	P	N	N	P	P	N	N	N	N	N	Y
	Architecture Complexity	36	14	9	36	27	?	4	32	?	?	50
	Structural(a) & Behavioral Models(b)	(a)	(a)	N	(a)	Y	N	N	N	N	N	Y
	Architectural Pattern	**A**	**B**	**C**	**D**	**E**	**F**	**F**	**G**	**H**	?	**I**
Variability Model Support	Integer Attributes	Y	N	Y	Y	N	N	N	Y	N	?	Y
	First Order Constraints	N	N	Y	P	N	N	Y	N	N	N	Y
	Context Aware Dynamic SPL	P	P	N	N	N	N	N	Y	N	N	Y
	Human Interpretable VM Semantic Language	P	N	N	N	N	N	N	P	N	N	Y
Solver Support	Integer Domains	P	P	Y	Y	N	N	N	Y	N	?	Y
	First Order Constraints	N	N	N	N	N	N	N	Y	N	?	Y
	Optimization	Y	?	Y	Y	Y	Y	?	Y	N	?	Y
	Meta-Programming	Y	Y	Y	P	Y	?	Y	?	Y	?	Y
	Incremental Solutions	Y	Y	Y	Y	Y	Y	Y	Y	Y	?	Y
Reasoning Task Spec Support	Declarative Verification Task Specification Language	N	N	P	N	Y	N	N	N	N	?	Y
	Declarative Configuration Task Specification Language	N	N	P	N	Y	N	N	N	N	?	Y
Standards	Leverages Existing Standards	N	N	N	P	N	N	N	N	N	?	Y

Tools: **(1)** FeatureIDE; **(2)** FlamaPy; **(3)** FAMILIAR; **(4)** COFFEE; **(5)** Kernel Haven; **(6)** SPLOT; **(7)** Glencoe; **(8)** ClaferTools; **(9)** Pure::Variants; **(10)** Gears; **(11)** Our Approach
Architecural Patterns: [A] Java API + Eclipse Plugin; [B] CLI + Python API (Plugin-Based); [C] CLI + Java API + Eclipse Plugin; [D] 3-Layered Web Service; [E] 3 Layered Monolith with Plugin System; [F] Client/Server (With no further details); [G] Haskell API/Web Server + Web Client; [H] IDE Plugins (With no further details); [I] Multi Layer Python API/Web Server + Pivot Language
Legend: *Y* – Yes; *N* – No; *P* – Partial Fulfillment; *?* – Unknown/Unclear;

Haven [28], pure::variants [10], and Gears [30]. Though most of these tools also cover other use-cases such as modeling, visualization or code generation, we focus on the verification pipelines.

In Table 1 we present an analysis of the characteristics of these tools. We divide our analysis into five main dimensions:

- First, we examine the different approaches from a purely architectural perspective. Of interest are the following characteristics: whether the approaches are well documented with standard modeling languages like the UML; the detail and granularity of the models as a function of the quantity of elements they contain; whether the models are structural, behavioral, or both; and, finally, what general architectural pattern is applied for their tool. We also analyze the interaction with model editors as part of their architectural patterns related to requirement **REQ4**.
- Next, we analyze the support their respective approaches have for the diversity of variability modeling languages. This includes the treatment of integers, and first-order constraints among the elements of a variability model. Two other key factors are whether the languages support concepts of context-aware dynamic SPLs and whether the semantics of the variability models are expressed in a human-interpretable language, or whether they are implicit as part of the tools' internal pipelines. This is related to requirement **REQ1** and the types of languages supported by competing tools.
- We analyze in a similar way the support for different solver features among the tools. We must, nevertheless, highlight the fact that our analysis of these features relates to the capabilities of the solvers themselves and not necessarily as they are used within the tool. For instance, though FeatureIDE has a mechanism to handle numeric attributes in their models, and their solving backend, a modified SAT4J [9], it cannot run automated verification or configuration tasks on these aspects of the models. Among the characteristics we examine we find the treatment of integer domains; whether there are first order constraints; whether optimization can be run in addition to pure solving; whether there are meta-programming mechanisms to control the behavior of the solvers; and whether the solvers can present series of solutions without restarting

the search space exploration from scratch. This is tied to our requirement **REQ3** and the backend support offered by the different tools.

- We also analyze the support for declarative specifications of the reasoning tasks to be performed and whether this is declarative and user-specifiable. This is related to requirement **REQ2**.
- Finally, we analyze the use of internationally recognized standards as part of the proposals as part of our analysis tied to our design principle **DP6**.

From this one can conclude that all of the approaches cited follow an approach of transforming the variability model into an input for a constraint solver. (It is worth repeating that here the term "constraint" is to be understood in its general sense, thus including other families such as SAT solvers, SMT solvers, etc.) We, therefore, have integrated these ideas and seek to generalize them to provide a truly generic architecture to perform these tasks and, moreover, lift the veil on the details of how these transformations are performed. Notably, the characteristics of the competing approaches tied in directly to our formulation of the requirements for the architecture.

5.2 Domain Specific Languages in SPL

A key feature of our architecture is its agnosticity w.r.t. DSVMLs; while one can make use of Feature Models and all its derivatives, the use of DSLs as alternative VMLs has been well attested in the literature, whose fit as alternatives to feature models is highlighted in [12], with some of the first proposals aiming to integrate textual DSVMLs directly into artefacts [6]. This was further explored in a survey article [49] where all the possible combinations of DSLs and Feature Models (and variations thereof) were explored, concluding that they may coexist to different degrees in a project, or that one may outright replace the VM with a DSVML model. Though their focus was primarily on textual-based DSLs, other authors have found it important to construct Graphical DSL based proposals. In this vein, the work of Demuth et al. [19] is particularily important. They proposed a tool that would allow one to construct DSLs and their meta-models all as part of a single graphical tool. This allows the generation of artefacts from the models since all of their elements are ultimately mapped to a restricted subset of UML, though the analysis capabilities were quite limited.

There have been other approaches to use and integrate Graphical DSLs for Variability Modeling, such as [44], presenting a case study of a DSL for creating variants of robot software. Several approaches hinged on creating custom UML profiles [13, 52] to aid in the automated generation of websites. A large set of industrial case studies where Graphical DSLs for managing variability were employed has also been presented [46], highlighting the use of a commercial DSL development tool [45]. This tool has also been used for creating a graphical DSL for an automatic performance test generation approach based on a product line approach [8].

These approaches, however, are all held back by one fundamental limitation which is the need for an ad-hoc implementation to construct a formal representation (if they do so at all), which does not allow one to tailor the underlying reasoning to the particular characteristics of the DSL, as we have done with Sawyer et al.'s language [40], which, though originally a graphical DSL for DPSL without any code generation capabilities, we have brought directly into our approach as a key demonstrator of its capabilities.

6 Evaluation and Discussion

6.1 Requirements coverage

Given the novelty of our approach, and the mechanism employed, we begin first with an analysis of the coverage of our requirements by our approach. This analysis is informed both by our analysis of the architecture in Section 3 and the lessons learned from the prototype implementation.

REQ1 *Support low-cost extension of existing DSVMLs and addition of new DSVMLs:* Our approach is specifically designed to use a declarative specification of the modeling languages, and, within reason, any VML could be handled by our approach. We illustrate this variety by implementing and testing our implementation with two languages that differ greatly syntactically and semantically.

REQ2 *Support low-cost addition of new automated reasoning tasks to run on the VM:* Our architecture is designed to handle precisely this, with a view on exposing a large variety of operations to the user so that as new DSVMLs are created, it is simple to create the corresponding reasoning tasks. As a consequence, the prototype reflects this and has been demonstrated to be robust enough to support very different reasoning tasks through the same mechanism.

REQ3 *The architecture must be agnostic* w.r.t. *the logical KRL and IE paradigm used for automated reasoning on the VM, supporting low-cost addition of interoperability with solvers from different paradigms:* Given that multi-solver support has been a key design goal for the architecture, this is covered by the N-to-one-to-M translation approach through CLIF, so that there is no hard-coded bias towards any IE and allowing easy integration of new IEs.

REQ4 *The architecture must be agnostic* w.r.t. *the DSVML editor tool, accepting the VM as input data exportable from multiple popular editors:* Our architecture seeks to be as independent as possible from any particular modeling tool by relying on the declarative specifications of the languages to perform all processing instead of relying on a particular set of technologies.

Our prototype implementation has been developed and tested initially on the VariaMos [47] modeling framework which has been modified to allow for the declarative specifications needed for the prototype. Further integration with other tools will require determining how to add this functionality to them.

6.2 Limitations of the architecture and prototype

The architecture has one important omission that is worth discussing. The architecture has an underlying assumption that the models are graphical models with possibly a portion of the model being text (constraints in CLIF). This has an important effect on the abstract syntax and therefore interpretation of the models: it is not yet clear how to deal with purely textual VMLs that don't use a well-known data interchange format like JSON. The reason for this is simple. Textual VMLs with their own grammar and structure would need their parser to be integrated into the architecture. In addition to this, a more abstract, common representation of both the hybrid and textual models would need to be constructed such that it would serve the base for constructing the semantics. Nevertheless, this seems possible with some modifications to the architecture, with, in particular, a more capable input management system.

The prototype we present is limited in some key ways. We have developed the prototype utilizing the VariaMos modeling environment, which has the underlying assumption that all the graphical models are directed graphs where the nodes can be typed and have attributes. This is sufficiently general for the DSLs we've observed in the literature, though one could imagine DSLs that use a more complex underlying structure. We seek nevertheless to combat this limitation through the abstract syntax specification mechanism which has only been partially implemented in the current prototype, given the limitation outlined in the previous paragraph.

6.3 Threats to validity

In addition to the limitations above, our work is subject to internal and external threats to validity. In terms of **Internal Validity** our primary concern is how feasible our architecture is. To counter this threat, we have created a prototype that covers as closely as possible the proposed architecture. The converse of this is the correctness of the prototype, that we've endeavored to test extensively. We have sought, therefore, to demonstrate its capacity to provide the functionality envisioned in the architecture. Another further threat is tied to the correctness of our architectural design. We have utilized a standard and well-understood modeling language (UML) for its definition and have sought to collect feedback from colleagues and users of the tool, both from a developer's perspective and from an end-user perspective.

Now, in terms of **external validity**, we recognize that we can't provide guarantees on the exhaustiveness of the tools surveyed, more recent efforts that the authors are unaware of might have been proposed in the meantime. Nevertheless, we have sought to base our overview on a recent and well sourced survey of these very tools. An additional threat concerns our ability to manage purely textual languages within the architecture, as mentioned in the previous section. To combat this we have designed a flexible and modular architecture capable of being modified in this direction.

6.4 Future Work

To overcome our limitations, work continues in several directions: first, textual languages, beginning with UVL, are being integrated; second, work continues on enlarging the set of reasoning requests possible, including having a more fine grained control over the variables used for iteration; third, more output languages, beginning with the Z3 SMT solver, are being integrated and tested, to further guarantee the robustness of CLIF as pivot representation; and, finally, work is being done on adding an incremental solving component (as opposed to iterative) in order to make interactions far more efficient. This will all imply minor refinements of the architecture, though we believe the core will remain unchanged.

Another future work we envision is to treat the architecture itself as an SPL, such that we could create fully configurable distributions for particular needs, i.e. include commercial solvers if the licenses are present, and, more importantly, give a larger freedom on the query formulation. We see value in this direction because the decision system for DSPLs can be modeled and then exported for whatever solver is supported, or even combinations of solvers, which could enable portfolio [3] constraint solving using multiple engines.

7 Conclusion

In this paper, we have presented an innovative software architecture to provide automated reasoning for SPLE. As we have seen in section 5, its main contribution is to be the first proposal focused on being agnostic with respect to both the VML used to model variability, and the constraint solving paradigm used to implement the reasoning. Such agnosticism allows fully decoupling the VM and SPL asset editing tools from the automated reasoning tools. We have put forward four requirements for an architecture to achieve such agnosticism. We then have presented fairly detailed structural and behavioral models for the architecture and discussed why it satisfies those requirements. Its key ideas are (a) to avoid combinatorial explosion of components by using a pivot standard semantic language and (b) pass as parameters declarative specifications of (i) the abstract syntax and semantics of the VML used by the VM to analyze and (ii) the analysis task itself.

We have validated this architecture by quickly refining and instantiating these models allowing for four distinct SPLE reasoning pipelines already available in our PLEIADES

framework prototype. Each one combines any of two VML languages, respectively feature models and a context-aware dynamic SPL VML, with solvers from two logical reasoning paradigms, respectively CLP, through libraries of SWI-Prolog, and CSP, through the CSP solver integration layer MiniZinc. We hope that our demonstration of the feasibility of VML and solver agnostic SPLE reasoning services together with the simplicity of instantiating the PLEIADES architecture we propose into a working implementation, reusing various VML and solvers, will foster more reuse of third party SPL model editors and third party solvers in the SPLE community.

References

[1] 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.

[2] 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

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

[6] Don Batory, Clay Johnson, Bob MacDonald, and Dale Von Heeder. 2000. Achieving Extensibility through Product-Lines and Domain-Specific Languages: A Case Study. In *Software Reuse: Advances in Software Reusability*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, and William B. Frakes (Eds.). Vol. 1844. Springer Berlin Heidelberg, Berlin, Heidelberg, 117–136. https://doi.org/10.1007/978-3-540-44995-9_8

[7] 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

[8] Maicon Bernardino, Avelino F. Zorzo, and Elder M. Rodrigues. 2016. Canopus: A Domain-Specific Language for Modeling Performance Testing. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST).* IEEE, Chicago, IL, USA, 157–167. https://doi.org/10.1109/ICST.2016.13

[9] Daniel Le Berre and Anne Parrain. [n. d.]. The Sat4j Library, Release 2.2. ([n. d.]).

[10] 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

[11] 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

[12] 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

[13] Juan José Cadavid, Juan Bernardo Quintero, David Esteban Lopez, Jesus Andrés Hincapié, Antonio Brogi, Araújo João, and Raquel Anaya. 2009. A Domain Specific Language to Generate Web Applications.. In *CIbSE.* 139–144.

[14] 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.

[15] 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

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

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

[18] I. Dejanović, R. Vaderna, G. Milosavljević, and Ž. Vuković. 2017. TextX: A Python tool for Domain-Specific Languages implementation. *Knowledge-Based Systems* 115 (2017), 1–4. https://doi.org/10.1016/j.knosys.2016.10.023

[19] Andreas Demuth, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2011. Cross-Layer Modeler: A Tool for Flexible Multilevel Modeling with Consistency Checking. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering.* ACM, Szeged Hungary, 452–455. https://doi.org/10.1145/2025113.2025189

[20] Douglas C Engelbart. 1962. Augmenting Human Intellect: A Conceptual Framework. *Menlo Park, CA* 21 (1962).

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

[22] 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.

[23] Michael R Genesereth and Richard E Fikes. 1992. Knowledge Interchange Format-Version 3.0: Reference Manual. (1992).

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

[25] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2008. Dynamic Software Product Lines. *Computer* 41, 4 (April 2008), 93–95. https://doi.org/10.1109/MC.2008.123

[26] 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

[27] 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

[28] 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]

[29] CharlesW Krueger. 2001. Easing the transition to software mass customization. In *International Workshop on Software Product-Family Engineering.* Springer, 282–293.

[30] 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.

[31] Philippe Lalanda, Julie A McCann, and Ada Diaconescu. 2013. *Autonomic computing: principles, design and implementation*. Springer Science & Business Media.

[32] Shih-Hsi Liu. 2010. *Design Space Exploration for Distributed Real-Time*. VDM Verlag Dr. Müller.

[33] 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.

[34] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM* 3, 4 (1960), 184–195.

[35] 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

[36] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *Principles and Practice of Constraint Programming – CP 2007*, Christian Bessière (Ed.). Vol. 4741. Springer Berlin Heidelberg, Berlin, Heidelberg, 529–543. https://doi.org/10.1007/978-3-540-74970-7_38

[37] 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.

[38] 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.

[39] Camille Salinesi and Ral Mazo. 2012. Defects in Product Line Models and How to Identify Them. In *Software Product Line - Advanced Topic*, Abdelrahman Elfaki (Ed.). InTech. https://doi.org/10.5772/35662

[40] Pete Sawyer, Raul Mazo, Daniel Diaz, Camille Salinesi, and Danny Hughes. 2012. Using Constraint Programming to Manage Configurations in Self-Adaptive Systems. *Computer* 45, 10 (2012), 56–63.

[41] DC Schmidt. 2006. Model-Driven Engineering. *Computer* 39, 2 (2006), 25–31.

[42] 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.

[43] 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

[44] Susumu Tokumoto. 2010. Product Line Development Using Multiple Domain Specific Languages in Embedded Systems. (2010).

[45] Juha-Pekka Tolvanen and Steven Kelly. 2018. Describing Variability with Domain-Specific Languages and Models. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*. ACM, Gothenburg Sweden, 300–300. https://doi.org/10.1145/3233027.3233059

[46] Juha-Pekka Tolvanen and Steven Kelly. 2019. How Domain-Specific Modeling Languages Address Variability in Product Line Development: Investigation of 23 Cases. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 155–163. https://doi.org/10.1145/3336294.3336316

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

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

[49] 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

[50] Danny Weyns. 2020. *An introduction to self-adaptive systems: A contemporary software engineering perspective*. John Wiley & Sons.

[51] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2010. SWI-Prolog. arXiv:1011.5332 [cs]

[52] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. 2004. Towards a UML Profile for Software Product Lines. In *Software Product-Family Engineering*, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, and Frank J. van der Linden (Eds.). Vol. 3014. Springer Berlin Heidelberg, Berlin, Heidelberg, 129–139. https://doi.org/10.1007/978-3-540-24667-1_10

Received 7 July 2023; revised 10 September 2023; accepted XX YYYY ZZZZ