A Lightweight Method to Define Solver-Agnostic Semantics of Domain Specific Languages for Software Product Line Variability Models

Camilo Correa Restrepo
Centre de Recherche en Informatique (CRI)
University of Paris 1 Panthéon-Sorbonne
Paris, France
email: camilo.correa-restrepo@univ-paris1.fr

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

Andres López
Investigación y desarrollo
SoftControlWeb
Medellin, Colombia
email: andresorlandolopez@gmail.com

Jacques Robin
Learning, Data and Robotics Laboratory, ESIEA, Paris, France
Center for Research in Informatics (CRI), University of Paris 1 Panthéon-Sorbonne, Paris, France
email: jacques.robin@esiea.fr

***Abstract*—We propose a method to address the current lack of standards for both software product line variability modeling languages and their formal semantics. It allows specifying, in an agile, declarative, and solver-agnostic fashion the formal semantics of a domain-specific variability modeling language through a simple JSON based specification format. Our approach leverages the Common Logic Interchange Format (CLIF) standard for interoperability among logical inference engines. We demonstrate our approach with two concrete examples of Variability Models, and present the tooling and architecture that makes this possible.**

Keywords*—*Variability Modeling; Formal Semantics; Modeling Language Specification; Common Logic.

I. Introduction

Software Product Lines Engineering (SPLE) [1] is a method to systematically coordinate and automate the engineering and evolution of a large set of related software products with overlapping functionalities and reusable software assets over long life-cycles, whose products, put together, form a ***Software Product Line (SPL)***. In model-driven SPLE, these assets are both models and code files while, in code-driven SPLE, these assets are mostly code files implementing services, components, classes, decorators, aspects and functions. The key artifact that distinguishes an SPL from a single software product is its ***Variability Model (VM)***. It explicitly identifies sets of requirements, generally called *features*, that are cohesive from a business or technical perspective and determines how they partially overlap across the different products of the line. This VM generally organizes those features into an abstraction and composition hierarchy and associates the lowest level ones with reusable and composable concrete software assets implementing them. Developing an SPL VM and such composable assets requires a large upfront investment. However, once done, it enables the automated generation of a very large number of product variants, which, in turn, supports the simultaneously low-cost and rapid delivery of very many customized software products, all maintained and evolved in coordination. It has provided great returns on investment mostly within industries such as transportation, healthcare and energy [2] where systems have a long lifecycle and have a critical nature.

Initially, an SPL VM was a purely design-time artifact used to interactively choose a valid set of features and attribute value choices to then generate the source code of a product variant (*a.k.a.* an SPL configuration) resulting from these choices by composing and/or transforming the associated SPL's reusable assets. More recently, they have started to be used as ***Models-at-Run-Time (M@RT)*** [3] artifacts for context-aware self-adaptive systems that continuously monitor their execution context for changes that might require a run-time reconfiguration. In this approach, called *dynamic* SPLE [4], an additional context model needs to be included into the SPL VM and the whole SPL and its configuration tool are embedded into each deployed product they generate. When monitoring systems detect that in a new context, the current configuration no longer satisfies some system requirements, it triggers the SPL configuration tool to search the SPL VM for alternative configurations better adapted to this new context and then update the implementation with it. To contrast them from dynamic SPLs, the original, purely design-time SPLs are called *static* SPLs.

In the current state of the art, there is no accepted standard for SPL VMs, so every SPLE tool uses its own ***Domain Specific Language (DSL)*** to model the VM. Nonetheless, almost all of these languages used for VMs share four key expressive capabilities. The first is to distinguish between mandatory and optional elements. The second is to specify ranges of alternative possible values for a given element parameter. The third is to specify ranges of alternative possibilities for the refinement of a higher-level elements into a set of lower-level elements. The fourth is to specify complex business and regulatory constraints concerning the co-occurrence of various elements or values across the abstraction hierarchy, that any product must satisfy, while also being implementable by a subset of the reusable assets available in the SPL. The existence of this common core results from the main purpose of any VM: supporting semi- or fully-automated configuration of a particular product out of the product line. Typically, this configuration process is divided into two stages. The first consists of choosing one valid point in the problem space represented by alternatives in the VM. The second consists of deriving a working implementation solution from the reusable assets associated with the options selected during the first stage.

As SPLs grow larger, VMs grow increasingly complex. Real-life industrial SPLs routinely contain over 10K elements and constraints. Since the problem and solution spaces are subtly but sparsely constrained combinations of the optional and alternative VM elements, their sizes are subject to a combinatorial explosion. This makes fully manual configuration impractical. It also inevitably leads to the introduction of inconsistent elements or constraints during the engineering and evolution of the VM. Therefore, as with any software artifact, the VM needs to be verified and validated with the help of automation tools.

A wide variety of approaches have been proposed to implement SPL VM verification and VM-guided SPL configuration tools. Just like for SPL VM languages, there is also currently no accepted standard API for such tools, though the overwhelming majority of them share a key feature in common: they rely on some form of *logical* knowledge representation and automated reasoning. This allows them to reuse practically scalable inference engines developed over the last 50 years by two research communities, the formal software engineering methods community and the artificial intelligence community. Four main classes of such engines have been extensively proposed and evaluated to automate SPL VM verification and VM-guided configurations: ***SATisfiability (SAT)*** solvers and their ***Satisfiability Modulo Theories (SMT)*** successors, ***Constraint Satisfaction Problem (CSP)*** solvers, ***Logic Programming (LP)*** engines and their ***Constraint LP (CLP)*** successors and ***Description Logic (DL)*** engines and their semantic web successors. No member of these engine families is a silver bullet for all SPL VM verification or VM-guided SPL configuration problems. They have subtle differences in expressiveness and performance on different kinds of problems, even if expressed in the same DSL VM language.

In this paper we propose a novel, light weight approach to bridge the gap between, on the one hand, the existing diversity and lack of standardization in VM languages, and, on the other hand, the existing diversity of logical languages that have been proposed to provide VM languages with formal semantics and are accepted as input by various classes of inference engines. Our approach is based on two key ideas. The first is to use a lightweight, declarative, textual syntax to specify both the concrete and abstract syntax of a VM DSL. This textual syntax is encoded both as Python objects from the Pydantic library [5] and as JSON files in the Open API web service standard [6]. It can be seen as a more agile alternative to the traditional Model-Driven Engineering [7] based on diagrammatic models, meta-models, and meta-meta-models. It is the subject of another publication under preparation. The second key idea, which is the focus of the present paper, is the proposal of the ***Common Logic Interchange Format (CLIF)*** [8] standard from the ***International Organization for Standardization (ISO)***, originally put forward to support interoperability among logical inference engines, to represent the formal semantics of any VM DSL in a solver-agnostic fashion. It starts from realizing that the four main classes of logical languages listed above and commonly used for VM verification and VM-guided SPL configuration are all essentially sub-languages of CLIF in terms of their expressiveness. In addition, CLIF is also the language used to define the formal semantics of the fUML [9] standard, the formal core of the Unified Modeling Language (UML) [10]. Therefore, any model-driven SPLE approach using the UML to model assets, could leverage the mapping from VM models to CLIF to provide a uniform formal semantics for the whole SPL model comprising both the VM and the asset model.

The main contribution of this paper is to propose a first step towards a common formal semantics for SPL VM based on an ISO standard. We show, with a couple of illustrative examples, how the semantics of two very different SPL VM graphical languages, Extended Feature Models for static SPL VMs and Sawyer et al's [11] extension of the ***Knowledge Acquisition in autOmated Specification (KAOS)*** modeling language [12] for context-aware dynamic SPL VMs, can both be uniformly expressed in CLIF. We also describe the architecture of the VariaMos tool that validates the approach by allowing one to specify, in CLIF, the semantics of a SPL VM DSL and then automatically generate the CLIF formula to logically represent this semantics for a specific, graphically edited SPL VM. Given that CLIF is expressive enough to capture the restricted subsets of ***First Order Logic (FOL)*** accepted as input by most constraint solvers (which will be touched upon in the following section), this contribution will allow the subsequent use of a variety of inference engines that can be tailored to each VM language. We demonstrate our approach within an open-source tool called VariaMos [13] that allows its users to specify the concrete visual syntax, the abstract syntax and the formal semantics in agile, declarative, textual and uniform fashion as JSON files. The formal semantics JSON specification then serves to associate abstract syntax elements with CLIF elements formulas.

The rest of the paper is organized as follows: in Section II, we present an overview of the background and work related to our approach; in Section III, we present our proposal for the use of CLIF as the standard formal semantics for Variability Modeling; in Section IV, present our CLIF translation mechanism by example, by examining the translation of two different modeling languages; in Section V, we elaborate on our use of CLIF and the specific dialect we have chosen; in Section VI, we present the overall architecture of our prototype and its implementation; in Section VII, we cover the limitations of our approach and outline planned future work; and, finally, in Section VIII, we present our conclusions.

II. Background and Related Work

There have been many approaches to establishing formal semantics for variability modeling languages; these have generally always been defined in the context of performing automated analyses of the constructed models. Since the constructs for each language vary, the corresponding semantics have always been defined as a function of the expressiveness of each language. Some of the first exploratory works on this topic proposed the use of first-order logic to provide the semantics for ***Basic Feature Models (BFMs)*** [14] (the simplest and original type of VM), though they essentially remained within the propositional core of FOL and only needed first order constructs to encode their semantics into manually constructed Prolog programs. As BFMs evolved, so too did their semantics, and, in particular, Benavides et al. [15] provided a characterization of Cardinality-based feature models as Satisfiability [16], Binary Decision Diagram [17] and (Boolean) Constraint Satisfaction Problems [18], all falling into the purview of first order theories.

There exist many variability modeling languages that are used for SPLE and beyond. The models constructed with these languages aim to capture the variability relations that exist within a given domain with the aim of expressing the set of allowable combinations of domain elements in products. These domain elements are commonly modeled as "features" that encode an end-user-facing piece of functionality [1]. The relations among these features make explicit the design constraints imposed both by the domain itself and the technological choices involved. There has been a considerable amount of work regarding the automated analysis of these models during the past few decades [19], such as automated configuration of products or finding errors in the models. These efforts primarily focused on models of a particular type, that is, feature models, which were originally proposed in [20] and have been since extended with additional constructs that increase their expressivity. A considerable amount of alternative modeling languages, and even syntactic variations of the aforementioned variability models have been proposed, each aiming to improve upon the characteristics of these feature models to support more expressive models that better capture the nature of the domain.

It has been noted in the literature that (finite domain) constraint solving approaches are those best suited to handle the expressivity of features models extended with numerical and symbolic constructs as surveyed by Benavides et al. [21]. This survey highlights several other semantic approaches that have been proposed in the literature, like the use of Description Logic [22] originally proposed by Wang et al. [23]. That being said, the overwhelming majority of approaches fit squarely in the realm of classical predicate (first order) logic.

In addition, most of the works here cited, and cited in the above surveys [19] [21], demonstrate that the approaches, whenever constructed to support tooling, transform the variability models directly into the representations amenable for analysis by the underlying solver technologies. This makes these formalizations difficult to reuse, compare, debug and render them inflexible to changes in the input language. We therefore diverge from these approaches and aim to construct a representation that directly encodes first order formulas, i.e., Common Logic [8], and in particular its machine- and human-interpretable syntax, the Common Logic Interchange Format or CLIF.

III. CLIF as standard formal semantics for Variability Modeling

One of the main contributions of this article is the proposal of the Common Logic [8] standard as the ideal target representation of the logical semantics of variability models. In particular, we propose the use of a fully conformant subset of the Common Logic Interchange Format (CLIF) as the preferred notation towards which transformation procedures should aim to produce their results. While CLIF's expressivity surpasses that of First Order Logic (FOL) through some additional constructs allowing for infinite expressions, we consider that there is only need to support the constructs effectively contained in FOL (c.f. Section 6.5 of [8] for a deeper justification and discussion as to why this is admissible and does not fundamentally limit our expressiveness). The justifications for choosing Common Logic (and CLIF in particular) are threefold: first, its capacity to be as expressive as FOL means that it easily represents all constructs that are handled by the most commonly used tools for analysis [19] [21], namely Constraint Logic Programming over finite domains [24], constraint programming [18], SAT solvers [16] and SMT [25] solvers, among others; second, its status as an international standard with a normative and fully defined representation format (CLIF) makes it easily interoperable with other systems and understandable by humans and machines alike; and, finally, the Lisp-like S-expression derived syntax make parsing and managing models represented in CLIF simple. In addition, this same structure facilitates the generation of these expressions from model elements.

There is an additional angle to consider as to why CLIF is particularly suitable for providing the semantics of models. Real world SPL projects go beyond domain VMs, and model the concrete software assets or artifacts that are to be used to assemble software products. UML models model the structure and behaviour of software systems, and are one possible type of asset model. As highlighted in the introduction, there have been ongoing efforts to provide formal semantics for UML models for automated execution and analysis, which have been defined in CLIF [9] for a subset of UML models. This opens the door to a possible avenue for investigating the logical integration between variability models and UML-derived asset models within a single analysis framework. CLIF has also found use in other domains, such as the basis for a large

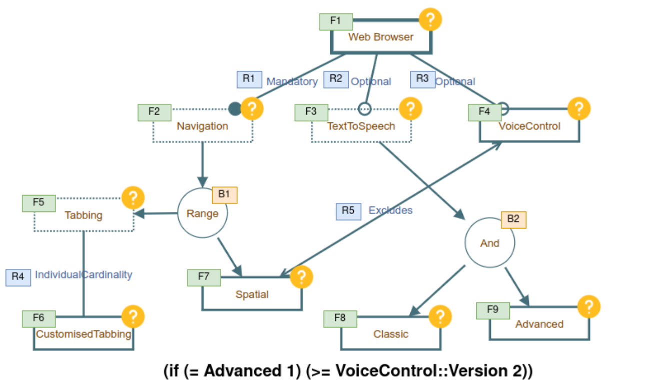

Fig. 1. An extended feature model with an arbitrary cross tree constraint depicted in VariaMos. Adapated from the example in Figure 2 in [29].

```
(model                                                              1
    (and (bool Web Browser) (= Web Browser 1))                      2
    (bool Navigation)                                               3
    (bool TextToSpeech)                                             4
    (bool VoiceControl)                                             5
    (int VoiceControl::Version)                                     6
    (bool Tabbing)                                                  7
    (bool Spatial)                                                  8
    (and (=< (Navigation * 1) (Tabbing + Spatial))                  9
        (=< (Tabbing + Spatial) (Navigation * 2)) )                 10
    (and (int (0 5) CustomisedTabbing) (and                         11
        (=< (Tabbing * 1) CustomisedTabbing)                        12
        (=< CustomisedTabbing (Tabbing * 5)) ))                     13
    (= (Classic + Advanced) (TextToSpeech * 2))                     14
    (bool Classic)                                                  15
    (bool Advanced)                                                 16
    (= Web Browser Navigation)                                      17
    (>= Web Browser TextToSpeech)                                   18
    (>= Web Browser VoiceControl)                                   19
    (=< (Spatial + VoiceControl) 1)                                 20
    (if (= Advanced 1) (>= VoiceControl::Version 2))                21
)                                                                   22
```

Fig. 2. The logical semantics of the feature model from Figure 1 in CLIF.

repository of formal ontologies [26], or as the input language for a tool that brings together heterogenous theorem provers [27].

IV. Illustrating VM language agnosticism by example

In this section, we aim to demonstrate the genericity of our approach by examining the translation of two different VM languages into CLIF. The key idea behind both of these examples is that, by providing each VM language with a specification for its semantics, we can transform any model constructed with said language into its corresponding logical semantics. These semantics could, in turn, be used as the input for inference engines with which different analyses could be performed. To achieve this, we make use of JSON [28] specifications that act as sets of templates for each of the elements present in each language's abstract syntax. These templates generate logic formulas with CLIF syntax, and, when collected together (with an implicit conjunction of all these formulas), form a complete CLIF model, that acts as the logical theory one associates to a given model.

To capture as large a gammut as possible of VM languages, we allow the semantics to be defined for all syntactic constructs that can be depicted in our modeling tool. In addition, some languages include constructs that reify, for instance, one-to-many relations that, to render their semantics, need information from neighboring nodes in the graph; therefore, we explicitly allow translation rules to capture information about neighboring elements in the graph to generate the CLIF formulas.

A. Feature Models

Figure 1 depicts an extended feature model for a product line of accesible web browsers featuring cardinalities and attributes. All the elements of the model have been annotated and numbered according to their type: features are in green; relations in blue; and elements that reify one to many relations (with UML-like cardinality ranges), called bundles, are in light orange. The logical semantics of the model in CLIF are portrayed in Figure 2. The correspondence between the model and its semantics is as follows:

- Line 2 depicts the semantics of F1 as a boolean variable, and, since it is a root feature, it also models a constraint that obligates it to be present, i.e., set to 1.
- Lines 3–5, 7–8, and 15–16 represent features F2–5 and F7–9 as boolean variables.
- Line 6 represents the Version attribute (not visible in the figure) of F4 as an integer variable.
- Line 9 represents the semantics of the reified relation B1, giving a range of 1 to 2 selected features if feature F2 is present.
- Line 11 represents the semantics of F6, a feature that, unlike the others, is not boolean, but can instead be present as up to 5 instances (also called clones in the literature).
- Line 14 represents the reified "and" relation B2, implying that if the parent feature F3 is present both F8 and F9 must be present aswell.
- Line 17 represents the mandatory relation R1, i.e., the two features must be bound to the same value.
- Lines 18 and 19 represent the optional relations from F1 to F3 and F4.
- Line 20 encodes the exclusion relation between F7 and F4.
- Line 21 encodes a complex constraint between F9 and the Version attribute of F4.

The transformation of the model and its elements is done through the specification of the semantics of the VM language's syntax elements in a JSON format. This JSON serves to provide a set of "templates" to turn these abstract syntax elements of the model into CLIF expressions. Figure 3 presents a fragment of the translation rules necessary to transform a feature model into its corresponding CLIF model. These templates form a bridge between the graph structure of the model and the CLIF expressions that represent them:

- Lines 1–7 define the semantics of the (boolean) features

```
1  {
2    "elementTypes": ["ConcreteFeature", ...],
3    "elementTranslationRules": {
4      "ConcreteFeature": {
5        "param": "F", "constraint": "(bool F)", ...
6      }
7    },
8    "relationTypes": ["Excludes", ...],
9    "relationPropertySchema": {
10     "type": { "index": 0, "key": "value" }
11   },
12   "relationTranslationRules": { ...,
13     "Excludes": { "params": ["FA","FB"],
14       "constraint": "(=< (F1 + F2) 1)"
15     },
16     "Optional": { "params": ["FA","FB"],
17       "constraint": "(>= F1 F2)"
18     },
19     "Mandatory": { "params": ["F1","F2"],
20       "constraint": "(= F1 F2)"
21     }
22   },
23   "relationReificationTypes": ["Bundle"],
24   "relationReificationTranslationRules":{
25     "Bundle": { "param": ["F","Xs","min","max"],
26       "paramMapping": {
27         "inboundEdges": {"unique": true,"var": "F"},
28         "outboundEdges": {"unique": false,"var": "Xs"}
29       },
30       "constraint": { ..., "And": "(= (sum(Xs)) (F * len(Xs)))",
31         "Range": "(and (=< (F*min) (sum(Xs))) (=< (sum(Xs)) (F*max)))"
32       }
33     }
34   },
35   ...
36 }
```

Fig. 3. Fragment of Feature Model Semantic Translation specification JSON.

as boolean variables where where *bool* is a distinguished unary predicate defining the domain of the variable F, i.e., $F \in \{0, 1\}$.

- Lines 8–21 define the semantics of the relations between features as CLIF expressions with arithmetic predicates and the mechanism for determining their type according to their properties.
- Lines 23–34 define the semantics of the bundles, and, given their one-to-many nature, define which of the two sides (ingoing or outgoing) contains multiple edges to expand expressions like *sum* or use their length in the CLIF expressions. In addition, depending on the type of the bundle, additional properties of the node may play a role, like the min/max properties for a range.

B. Sawyer et al.'s Variability Modeling Langauge

Figure 4 depicts a fragment of the principal model, using a modified KAOS [12] language, proposed in [11]. It models a flood early-warning system and how the system can modify its operational configuration depending on the state of its environment as reported through sensors. The language they have defined is structured as follows:

- Goals, with labels in green, determine the functional requirements of the system and are analogous to features in feature models. Goals form a hierarchy wherein the lower level goals imply how the goals they point to are to be achieved.
- Soft Goals, depicted as clouds annotated in Blue, encode the non-functional requirements of the system and can be satisfied in a 0 to 4 scale, which is encoded as "--", "-", "=", "+", "++" in the model. They themselves form a hierarchy in a manner analogous to goals.
- Context Variables, annotated in light red, encode the state of the system's environment among a set possible choices enconded as an enumeration of strings.
- Operationalizations, labeled in gray, specify the ways in which a goal can be satisfied and correspond to concrete modes of operation of the system. They are tied to a goal through Bundles, in light orange, which behave analogously to feature models (though with the edge direction reversed).
- Claims, annotated in magenta, express the level to which operationalizations satisfy the Soft Goals as a function of which has been selected.
- Soft Influences, labeled in yellow, 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 aim with this language is to construct an optimization problem where the largest amount of claims can be satisfied in terms of the selected operationalizations, and therefore ensure that the Soft Goals are satisfied to a given level.

As before, we can interpret the semantics interpretation of the CLIF model in Figure 5 as follows:

- Lines 2–4, 11–14 encode the the Goals G1–3 and the operationalizations O1–4 as boolean variables. In addition, lines 45 and 46 encode the relations of the subgoals to the main goal.
- Lines 5–10 encode the value relations of the Softgoals with those above them in the hierarchy as their average.
- Lines 15–18 encode the bundles B1–2 as the choices between the operationalizations.
- Lines 19–30 encode the consequences of the claims C1–4 on the Soft Goals, with the claims themselves being a boolean variable that is true iff their claims are satisfied in the resulting configuration.
- Lines 31–32, 35–40 encode the semantics of the Soft Influences on the Soft Goals, and, just like the claims, are boolean values that are true iff the requirements are satisfied.
- Lines 33–34 encode the Context Variables CV1–2 as enumerations in a fixed range.
- Lines 41–44 encode the Soft Goals as bounded integer variables.

In Section V, we will continue with the analysis of this example, its semantic specification and the implications for representing VMs in CLIF.

V. Differences between the dialect used for semantic specification and (standard) CLIF

As was hinted at in the previous section, while we target CLIF as our representation, there are some practical consid-

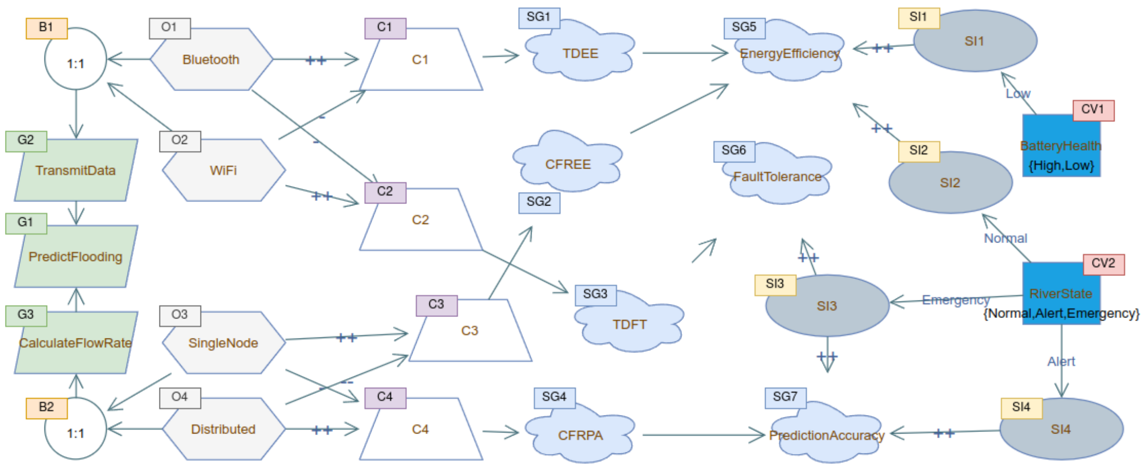

Fig. 4. A fragment of the model for the run-time variability of a flood early-warning originally proposed in and using the language of Sawyer et al.'s [11].

```
(model
    (bool PredictFlooding)
    (bool TransmitData)
    (bool CalculateFlowRate)
    (and (int (0 4) EnergyEfficiency)
        (= EnergyEfficiency ((CFREE + TDEE )/2)))
    (and (int (0 4) FaultTolerance)
        (= FaultTolerance ((TDFT)/1)))
    (and (int (0 4) PredictionAccuracy)
        (= PredictionAccuracy ((CFRPA)/1)))
    (bool Bluetooth)
    (bool WiFi)
    (bool Distributed)
    (bool SingleNode)
    (and (=< (TransmitData * 1) (Bluetooth + WiFi))
        (=< (Bluetooth + WiFi) (TransmitData * 1)))
    (and (=< (CalculateFlowRate * 1) (SingleNode + Distributed))
        (=< (SingleNode + Distributed) (CalculateFlowRate * 1)))
    (and (bool C1) (iff (= C1 1) (and
        (if (= Bluetooth 1) (=< TDEE 4))
        (if (= WiFi 1) (=< TDEE 1)))))
    (and (bool C2) (iff (= C2 1) (and
        (if (= Bluetooth 1) (=< TDFT 1))
        (if (= WiFi 1) (=< TDFT 4) ))))
    (and (bool C3) (iff (= C3 1) (and
        (if (= SingleNode 1) (=< CFREE 4))
        (if (= Distributed 1) (=< CFREE 0)))))
    (and (bool C4) (iff (= C4 1) (and
        (if (= SingleNode 1) (=< CFRPA 1))
        (if (= Distributed 1) (=< CFRPA 4)))))
    (and (bool SI1) (iff (= SI1 1) (if (= BatteryHealth 0)
        (and (= EnergyEfficiency 4)))))
    (enum (0 1 2) BatteryHealth)
    (enum (0 1) RiverState)
    (and (bool SI2) (iff (= SI2 1)
        (if (= RiverState 0) (and (= EnergyEfficiency 4)))))
    (and (bool SI3) (iff (= SI3 1) (if (= RiverState 2)
        (and (= FaultTolerance 4) (= PredictionAccuracy 4)) ) ) )
    (and (bool SI4) (iff (= SI4 1)
        (if (= RiverState 1) (and (= PredictionAccuracy 4)))))
    (int (0 4) TDEE)
    (int (0 4) CFREE)
    (int (0 4) TDFT)
    (int (0 4) CFRPA)
    (= TransmitData PredictFlooding)
    (= CalculateFlowRate PredictFlooding)
)
```

Fig. 5. The logical semantics of the feature model from Figure 1 in CLIF.

erations for its use that mean that we differ from CLIF as presented in the standard. These deviations, though small, have important consequences for the models produced through the semantic specification mechanism. The first of these departures is that we expressly recognize a set of distinguished predicates beyond the sole equality recognized by CLIF, such as unary or binary predicates like *int and bool* relating to the domains of variables used in the program. This is motivated by a desire to facilitate the construction of executable representations in different solvers that generally cannot automatically infer domain membership or require it outright for every variable. The other quite salient departure from the standard presentation of CLIF relates to the treatment of quantifiers; in effect, to enable complex arbitrary constraints, or even merely support the conjunction of the sentences outlined in the example from subsection IV-B, the reocurrence of the same variables must imply that they refer to the same object. Therefore, variables that occur free in the generated semantics are to be interpreted as being implicitly universally quantified over the conjunction of all the generated sentences; we do this following the tradition set by the standard first-order semantics for (pure) Prolog programs given by Clark's completion [30] and that of the ***Knowledge Interchange Format*** [31] language from which CLIF itself was derived. For example, the full logical reading of the exclusion relation in Figure 1 and therefore lines 5,8 and 20 from Figure 2 would be (using their annotated names for brevity):

$$\forall F_4, F_7 \bigwedge \{(F_4 \in \{0,1\}), (F_7 \in \{0,1\}), (F_4 + F_7 \leq 1)\}$$

This naturally leads to the question of handling quantifiers within each of the sentences. We have not yet found it necessary to introduce existential quantifiers for the semantics of the models languages we have dealt with so far; however, we have found very important applications of universal quantifiers, in particular, as a construct allowing one to deal with sentences that involve sets of incoming or outgoing edges to a given node in the directed graph. We currently consider, then, that the quantifiers range over finite sets that are defined precisely by the multiplicity of possible connections. This, in turn, leads to an important transformation that can be done directly without any loss of expressivity, namely that one can transform the universal quantification over a single variable and over a sentence into the iterated conjunction of a set of sentences each being the original sentence with the bound variable replaced by a member of the set. This is due to the following equivalence:

Let S be a finite set and ϕ an arbitrary first order sentence with only s unbound.

$$\forall s \in S \phi(s) \Leftrightarrow \bigwedge_{s \in S} \phi(s)$$

This greatly simplifies, and augments the power of, the semantics we can express and in turn greatly simplifies the translation from CLIF towards some solver paradigms where there is no pure or declarative handling of classical universal quantification, e.g., Prolog. To be clear, the quantifier expansion draws its values from the graph's elements and not from the variable domains over which we are solving. This subtle point means that we retain the intensive expression of the semantics but are able to write more general rules that have potentially varying amounts of appearances of certain elements from neighboring elements.

To demonstrate these mechanisms and differences in practice, consider the example from Figure 4, and in particular centered on the element SI3 and its relation with its neighbors SG6, SG7 and CV2. We have defined the semantics of the Soft Influence as shown in Figure 6. Within this semantic specification, we "bind" the set over which the x variable is quantified as the target nodes of the outbound edges, corresponding to X_s in the translation rule and to SG_6 and SG_7 in Figure 4. We also have distinguished functions relating to intrinsic properties of the directed graph, such as $edge(x)$ whose value is the edge leading to the x node, and we also allow references to arbitrary custom properties defined on graph elements through the :: operator.

The corresponding logical reading of these semantics would be as follows:

Let X be the set of nodes in the graph, $X_s \subset X$ be the set of nodes corresponding to the outgoing edges in the model, E be the set of edges in the model, $edge$ be a function $edge : X \rightarrow E$, V the set of attribute types and V_s the set of attribute values, :: be an infix binary function $(::) : E \cup X \times V \rightarrow V_s$, S be the variable corresponding to the id of the soft influence node, and $F \in \{0, 1\}$ the variable corresponding to the id of the unique inbound node. It is to be understood that the predicate imposing bounds on the F variable would also be part of the semantics.

$$\forall S, F\big[bool(S) \wedge \big((S = 1) \iff (F = edge(F) :: Value \implies \forall x \in X_s(x = edge(x) :: SatisfactionLevel))\big)\big]$$

However, given the equivalence cited above, the rendered semantics would be (lines 37–38):

```
{ ...,
  "relationReificationTranslationRules": {
    "SoftInfluence": {
      "param": ["S", "F", "Xs"],
      "paramMapping": {
        "node": "S",
        "inboundEdges": { "unique": true, "var": "F" },
        "outboundEdges": { "unique": false, "var": "Xs" }
      },
      "constraint": {
        "SoftInfluence": "\
        (and (bool S) (iff \
          (= S 1) (if \
            (= F edge(F)::Value)
            (forall (x:Xs) \
              (= x edge(x)::SatisfactionLevel) \
            ) \
          ) \
        ) )"
      }
    }, ...
  }, ...
}
```

Fig. 6. Fragment of Sawyer et al.'s language's Semantic Translation specification JSON. The "\" indicates the split in the multiline string for readability and formatting.

```
(and (bool SI3) (iff (= SI3 1)
  (if (= CV2 edge(CV2)::Value) (and
    (= SG6 edge(SG6)::Value) (= SG7 edge(SG7)::Value)))))
```

This removes the need to handle the internal quantifier, and simplifies the reading of the rendered formula.

VI. Variability Model to CLIF Translation Architecture and Implementation

We propose a distributed architecture for CLIF semantic translation, as depicted in Figure 7. The necessary tooling for modeling is served from a cloud infrastructure avoiding the need to perform any installation on the client beyond browsing to the website, where the user is served the user interface (shown in green) by the FrontEnd HTTP server. Within the cloud infrastructure, all concerns relating to the storage of the languages are handled, including their syntax and semantics, with a database backing these operations and providing a source of persistence between client interactions. An additional service that can perform certain checks on graph validity, such as allowed element amounts and connections beyond what can be handled natively by the graphics library used by the client is proposed in the architecture, to offload some responsibilities from the client. All of these services are deployed as containers, and are shown in white in our logical architecture diagram.

When it comes to the semantic translator service (shown in yellow), which is the core tool covered by this articles, it has been developed as a Docker [32] container which will expose a REST API endpoint over which the translation (and eventual analysis) operations will be served. Since our Front End is inherently configurable, this Back End can be deployed anywhere. For our prototype we have deployed it locally within a test cluster, but it will be made available within the

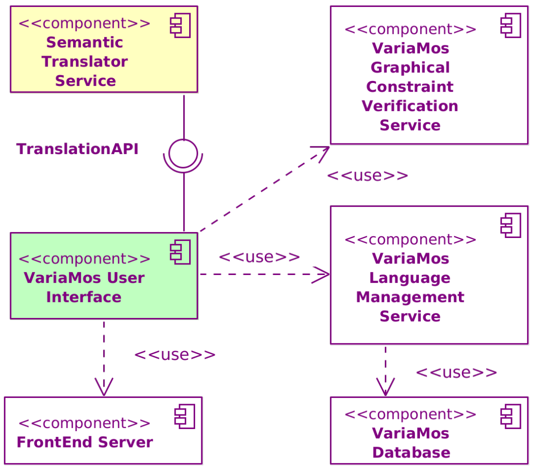
<<component>> Semantic Translator Service
TranslationAPI
<<component>> VariaMos Graphical Constraint Verification Service
<<use>>
<<component>> VariaMos User Interface
<<use>>
<<component>> VariaMos Language Management Service
<<use>>
<<use>>
<<component>> FrontEnd Server
<<component>> VariaMos Database

Fig. 7. High Level logical architecture of the VariaMos tool and the translation tool.

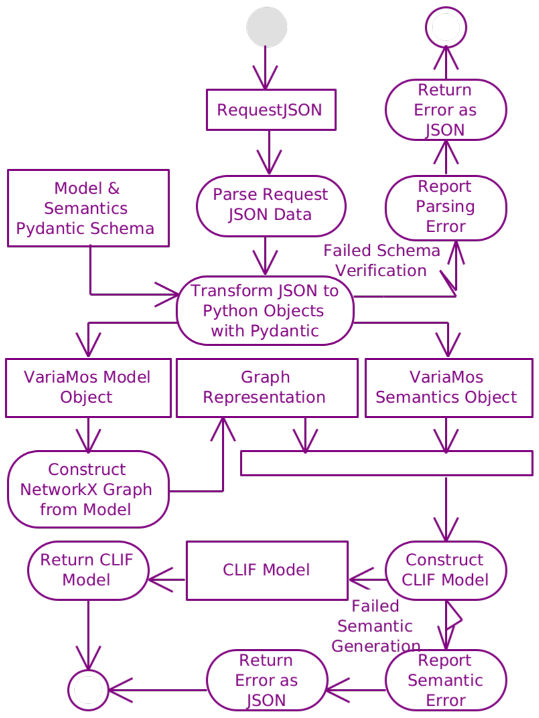

Fig. 8. High-Level operation of the translation tool.

VariaMos cloud infrastructure, where it will be the default point of access for translation. Looking forward, however, to the integration of the underlying solvers (which may also require licenses and hence can only be run locally), it seems clear that decoupling the translation mechanism allows for the best use of the available computational resources by allowing the user to choose where he desires the operations to be run, either sharing a cloud resource with others or deploying a local version of the container if he so wishes. This retains all the benefits of a cloud-native solution, while also being flexible when additional computational resources are required for a particular project.

The high-level operating principles of the translation mechanism are described in Figure 8. The fundamental operations carried out involve performing data validation on both the form of the provided Model and the semantics. Then the serialized model is reconstructed into a Graph; this, put together with the semantics, are then put through the CLIF model generation procedure which ultimately outputs the model's semantics which are the reported back to the user. This is all exposed through the API endpoint served in the translator container.

In terms of the implementation of the translation Back End, we have constructed the server in python with the following software components: Flask [33] for the server code, pydantic [5] for data schema validation, networkX [34] for graph representation in python, and textX [35] for managing the CLIF grammar. All of the code for the translator is open-source and freely available on GitHub at [36]. Within this repository are included the full grammar for our subset of CLIF, and complete examples (including models, semantics and syntax) for basic and extended feature models, as well as for Sawyer et al.'s [11] modified version of KAOS.

VII. Limitations and Future Work

In terms of our approach's limitations, it must be noted that we do not cover the entire CLIF standard, and we have not yet found a need to construct a particular treatment of existential quantification within model's semantics. There are also some limitations tied to the translation mechanism for the generation of the semantics, namely, we require that the elements that participate in a given node's or edge's semantics be directly connected, and thus we cannot yet perform the transitive traversal of the graph for the generation of the semantics. We also have no semantic treatment for elements that can be nested. We also require the user to have some understanding of the internal structure of the graph representation in the front-end client in order to define rules for attribute lookup to, for example, perform type disambiguation when a given node or relation's semantics depends on types defined by these attributes. This means that while we believe our approach is expressive enough to handle most VM languages that have been covered in the literature, it is possible that some others posses constructs that aren't easily expressible or require the aspects of CLIF we have not yet covered, such as deeply nested negation. In addition, we are restricted to languages that form directed graphs, so we are unable to treat languages with more complex graph types like hypergraphs without some measure of reification. We also do not handle the translation of textual languages into CLIF.

As mentioned in Section VI, our principal aim is to continue expanding upon the already implemented aspects of the proposal in order to complete the full end-to-end cycle of model generation and subsequent automated analysis through the use of several solvers. We will initially target constraint solvers as these are the best attested in the literature [37] and have the most straightforward translation from CLIF into their respective representation, but we ultimately aim to support a larger range of first order logic-based automated analysis tools.

VIII. Conclusions

In this article, we have presented a proposal for the use of CLIF as the standard representation format for the semantics of Variability models. We also present and demonstrate a mechanism to specify formal semantics for variability modeling languages through a simple JSON based specification format.

Our mechanism leverages the user-friendliness of the JSON format with the ability to quickly construct one's needed (and formally defined) modeling language semantics in such a manner that it spares prospective users from the need to learn the specifics of the programming involved. It also enables the quick evolution of language's semantics with no modifications to the underlying tools.

We believe this will permit the transparent integration of multiple analysis methods and especially solver families through the construction of translation from CLIF into their respective syntaxes.

Acknowledgment

We would like to thank Mauricio Agudelo for his work on the architecture and development of VariaMos, Jairo Soto for his work on the deployment of the VariaMos cloud infrastructure, and Francisco Piedrahita for his work on an initial version of these ideas.

References

[1] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*, 1st ed. New York, NY: Springer, 2005.

[2] Systems and Software Product Line Conference, "Product Line Hall Of Fame," https://splc.net/fame.html, n.d., accessed: 2023-03-27.

[3] B. H. Cheng *et al.*, "Using models at runtime to address assurance for self-adaptive systems," *Models@ run. time: foundations, applications, and roadmaps*, pp. 101–136, 2014.

[4] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *Computer*, vol. 41, no. 4, pp. 93–95, 2008.

[5] Pydantic Services Inc., "Pydantic," https://pydantic.dev/, 2023, accessed: 2023-03-27.

[6] OpenAPI Initiative, "OpenAPI Specification," https://spec.openapis.org/oas/latest.html, 2021, accessed: 2023-03-27.

[7] M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven software engineering in practice*, ser. Synthesis lectures on software engineering. Morgan and Claypool, 2017.

[8] "Information Technology – Common Logic (CL) – A framework for a family of logic-based languages," International Organization for Standardization, Geneva, CH, Tech. Rep., Jul. 2018.

[9] "Semantics of a Foundational Subset for Executable UML Models (fUML), version 1.5," Object Management Group, Tech. Rep., Apr. 2021.

[10] S. Cook *et al.*, "Unified Modeling Language (UML), version 2.5.1," Object Management Group, Tech. Rep., Dec. 2017.

[11] P. Sawyer, R. Mazo, D. Diaz, C. Salinesi, and D. Hughes, "Using constraint programming to manage configurations in self-adaptive systems," *Computer*, vol. 45, no. 10, pp. 56–63, 2012.

[12] A. Van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software*. John Wiley & Sons, 2009.

[13] VariaMos Team, "VariaMos Framework," https://variamos.com/, 2023, accessed: 2023-03-27.

[14] M. Mannion, "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, 2002, pp. 176–187.

[15] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés, "A first step towards a framework for the automated analysis of feature models," *Proc. Managing Variability for Software Product Lines: Working With Variability Mechanisms*, pp. 39–47, 2006.

[16] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, "Satisfiability solvers," *Foundations of Artificial Intelligence*, vol. 3, pp. 89–134, 2008.

[17] R. Drechsler and D. Sieling, "Binary decision diagrams in theory and practice," *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 112–136, May 2001.

[18] R. Dechter and D. Cohen, *Constraint Processing*. Morgan Kaufmann, 2003.

[19] D. Benavides, "Variability Modelling and Analysis During 30 Years," in *From Software Engineering to Formal Methods and Tools, and Back: Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday*, ser. Lecture Notes in Computer Science, M. H. ter Beek, A. Fantechi, and L. Semini, Eds. Cham: Springer International Publishing, 2019, pp. 365–373.

[20] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study:," Defense Technical Information Center, Fort Belvoir, VA, Tech. Rep., Nov. 1990.

[21] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, Sep. 2010.

[22] F. Baader, D. Calvanese, D. McGuinness, P. Patel-Schneider, and D. Nardi, *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge university press, 2003.

[23] H. Wang, Y. F. Li, J. Sun, H. Zhang, and J. Pan, "A semantic web approach to feature modeling and verification," in *Workshop on Semantic Web Enabled Software Engineering (SWESE'05)*, 2005, p. 46.

[24] J. Jaffar and J.-L. Lassez, "Constraint logic programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1987, pp. 111–119.

[25] C. Barrett and C. Tinelli, "Satisfiability Modulo Theories," in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Cham: Springer International Publishing, 2018, pp. 305–343.

[26] Semantic Technologies Laboratory, "COLORE," http://stl.mie.utoronto.ca/colore/, n.d., accessed: 2023-03-27.

[27] T. Mossakowski, M. Codescu, O. Kutz, C. Lange, and M. Grüninger, "Proof support for common logic." in *ARQNL@ IJCAR*, 2014, pp. 42–58.

[28] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)," Internet Engineering Task Force, Request for Comments RFC 4627, Jul. 2006.

[29] J. Carbonnel, M. Huchard, and C. Nebut, "Towards complex product line variability modelling: Mining relationships from non-boolean descriptions," *Journal of Systems and Software*, vol. 156, pp. 341–360, Oct. 2019.

[30] J. W. Lloyd and J. W. Lloyd, *Foundations of Logic Programming*, 2nd ed., ser. Artificial Intelligence. Berlin Heidelberg: Springer, 1993.

[31] M. R. Genesereth and R. E. Fikes, "Knowledge interchange format-version 3.0: Reference manual," 1992.

[32] Docker Inc., "Docker Documentation," https://docs.docker.com/, 2023, accessed: 2023-03-27.

[33] The Pallets Projects, "Flask User's Guide," https://flask.palletsprojects.com/en/2.2.x/, n.d., accessed: 2023-03-27.

[34] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[35] I. Dejanović, R. Vaderna, G. Milosavljević, and Ž. Vuković, "TextX: A Python tool for Domain-Specific Languages implementation," *Knowledge-Based Systems*, vol. 115, pp. 1–4, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950705116304178

[36] C. Correa Restrepo, "Semantic Translator," https://github.com/ccr185/semantic_translator, 2023, accessed: 2023-03-27.

[37] M. Pol'la, A. Buccella, and A. Cechich, "Analysis of variability models: A systematic literature review," *Software and Systems Modeling*, vol. 20, no. 4, pp. 1043–1077, Aug. 2021.