



HAL
open science

Constraints: the Core of Product Line Engineering

Camille Salinesi, Raul Mazo, Olfa Djebbi, Daniel Diaz, Alberto Lora-Michiels

► **To cite this version:**

Camille Salinesi, Raul Mazo, Olfa Djebbi, Daniel Diaz, Alberto Lora-Michiels. Constraints: the Core of Product Line Engineering. Fifth IEEE International Conference on Research Challenges in Information Science (RCIS), May 2011, Guadeloupe, France. hal-00707544

HAL Id: hal-00707544

<https://hal.science/hal-00707544>

Submitted on 13 Jun 2012

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.

Constraints: the Core of Product Line Engineering

Camille Salinesi¹, Raul Mazo^{1,2}, Olfa Djebbi¹, Daniel Diaz¹, Alberto Lora-Michiels³

¹CRI, Panthéon Sorbonne University, Paris, France

²Ingeniería de Sistemas, University of Antioquia, Medellín, Colombia

³Baxter International Inc, Lessines-Belgium

{camille, daniel.diaz}@univ-paris1.fr, {raulmazo, albertoloram}@gmail.com, olfa.djebbi@malix.univ-paris1.fr

Abstract— Product line engineering is a reuse-driven development paradigm based on the management of variability, which was successfully applied in information systems engineering and other domains. A common way to represent variability is with variability models that describe artefacts, and the dependencies between their various inflexions. Constraint programming, and in particular Boolean constraint programming, has been used so far to support analysis of variability models such as Feature-Oriented Domain Analysis (FODA) and the like. This paper goes a step further by using constraint programming to specify product lines. The focus on variability, variation points or dependencies is switched to the concept of constraints that apply to variables. The paper shows that this approach is richer than the one based on dependencies. For instance, many constraints that were needed in the cases we explored cannot be specified with dependencies of existing product line modelling languages. The approach was implemented in a prototype tool, and its scalability explored with industry case studies. These experiments show that constraint programming encompasses existing product line modelling languages such as FODA or OVM (Orthogonal Variability Model) and opens way to new possibilities such as reasoning simultaneously with different models during domain or application engineering.

Keywords- constraints, product line specification, product line analysis, constraint programming

I. INTRODUCTION

Propositional logic has been used to deal with Product Line Engineering (PLE) [1, 2, 3, 5]. More recent works [4, 6, 28, 29] show the interest of the international community of PLE to represent and analyze Product Line (PL) models by using constraint programming. Indeed, an analogy can easily be drawn between features based PL models and Constraint Programming (CP) because both can be defined by a collection of variables and a collection of constraints that these variables should satisfy. In contrast, specifying a product line as a constraint program instead of a feature model [27] has two important advantages: the expressiveness and the direct automation. On the one hand, variables in CP can take values over boolean, integer, real or even complex domains (i.e., lists, arrays and trees) and not only boolean values as in Feature-Oriented Domain Analysis (FODA) models [27]. On the other hand, constraints in CP can be boolean, arithmetic, symbolic and reified, and not only boolean as in FODA models. Besides, PL models expressed as constraint programs can be directly executed and analyzed by off-the-shelf solvers. This latest property avoids problems

related to loss of information and misinterpretation when de PL model is transformed from its original formalism to an executable language. The loss of information can be of two types: loss of structural information and loss of semantic information. In the first case, we do not have the possibility to identify anomalies related to the structure neither derivate a product guided by the structure of the PL model, because the structural properties of the model are lost. In the second case, we lose information about the semantic of the model, e.g. the number of products that can be derived from the PL model or knowledge about the ability of the PL model to derive products.

Different kinds of constraint programs can be defined depending on the types of variables on which the reasoning applies [9]; each can be solved with a specific kind of solver:

- Boolean e.g. SAT [3,10], BDD [4,11], SMV [2,12],
- Integer e.g. GNU Prolog [13], CHOCO [5,7,14],
- Reals e.g. clp(R) [15],
- or even trees and lists, e.g. Prolog-III [16].

It has been shown that FODA models can be represented as boolean constraint programs through a series of boolean variables, where each variable corresponds to a feature [4, 7]. A configuration is then generated using a SAT solver, under the form of a value for each variable, where TRUE means that the product has the corresponding feature. However, our bibliography review showed that only a part of CP capabilities has been exploited so far. So far, only few approaches have dealt with integer CP (or finite domain CP) [4, 8, 22, 24, 29]. However, we observed that transforming feature models into integer CP provides many advantages including dealing with advanced characteristics of FODA dialects (such as attributes or feature cardinalities), specifying more complex requirements than select/de-select a feature, or make complex analyses and verifications [8, 22, 29]. Another observation is that most existing approaches consist in transforming existing PL models into CP. We believe that this approach hinders the full exploitation of the versatility of CP. *The idea in this paper is to explore the expressiveness of constraint programming to specify product line models and to support its automation and reasoning.* Our goal is twofold: (i) at the domain engineering level, to widen the power of expression of PL specifications and support domain level PL analysis, and (ii) at the application engineering level, to provide new analysis features.

Our research strategy to achieve this was the following: first, we explored the power of expression of integer CP by specifying a simple but realistic PL. This allowed us both to evaluate the feasibility of the approach, but also to explore

the analysis capability offered by constraint solvers supporting the chosen integer CP language. The approach was then discussed with PL experts of companies like ADN, Renault, Stago and Baxter. Besides, we developed a series of transformation strategies to specify FODA models [27], UML-based variability models [31, 32] and OVMs [21] using integer CP. Last, we explored an industrial case study to evaluate our approach.

Our main working hypothesis in this work was to choose a CP language that can be handled by a solver. In this respect, Object Constraint Language (OCL) was not considered as a relevant language in our experiments, even though OCL could be used to specify PL constraints. The reason of this decision is that even if OCL is a well known language to represent constraints, OCL rules are executed by an interpreter and not by a solver, losing, in the way, some reasoning capabilities important in the domain of product lines (e.g., to know if a product line model allows generating at least one product, or to know the number of valid products represented in the product line model).

The rest of the paper is structured as follows: the next section introduces a working example that is used in the rest of the paper to illustrate our approach. Section 3 describes the approach by presenting CP with a meta-model and the various kinds of constraints that can be used to specify a PL. Section 4 shows equivalences between our integer CP based PL modelling approach and the more traditional FODA, OVM and UML-based variability approaches. Section 5 reports our evaluation of the expected advantages. The concluding sections discuss related works and research directions.

II. WORKING EXAMPLE

Consider the hypothetical Movement Control System (MCS) of a car (Figure 1). The purpose of such a system is to assist drivers to park, help them detect obstacles and while running, control their own speed and trajectory as well as the safety distance from other vehicles. Movement control is performed mainly through sensors that capture position and speed. A simple scenario is that drivers command actuators, which activate sensors. Sensors can then return feedback to drivers. Feedback can be visual, audio or by vibration. The coordination of information flow is ensured by a processor, which is characterized by its internal memory. Sensors and actuators can incorporate an auto-test functionality that checks the components functioning. Actuators auto-test checks the functioning and the memory, while sensors auto-test checks the functioning, the consistency and the response time.

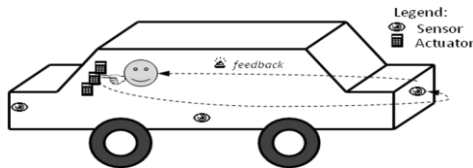


Figure 1. Overview of a movement control system

PL elements are listed below; they constitute the problem variables in which the domain indicates the allowed number

of occurrences, for example there can be 0 to 3 position sensors, and a memory can be of 4, 8, 16 or 32 MB:

- Position Sensor: (0..3)
- Distance Sensor: (0..4)
- Speed Sensor: (0,1)
- Actuator: (0..11)
- Processor: (0..4)
- Internal Memory: (4, 8, 16, 32)
- Visual feedback, Audio feedback, Vibration feedback: (0,1)

The MCS PL can be specified by the following constraints:

- C1.** A MCS system includes at least a sensor, an actuator and a processor.
- C2.** A MCS should contain at least one and at most two kinds of feedback (visual, audio, vibration).
- C3.** A MCS should contain an actuator for each instance of device in the following list: position sensors, front vehicle speed sensor, distance sensor, visual feedback, audio feedback, and vibration feedback.
- C4.** Sensor functionality check should be integrated to the MCS if it contains at least one kind of sensor (position, distance or front vehicle speed).
- C5.** Actuator functionality check should be integrated to the MCS if it contains at least one actuator.
- C6.** When a sensor consistency check is included in a configuration, then sensor response time check must be included too.
- C7.** The memory should be doubled each time a processor is added: the pair <processor; internal memory> can have the following values <1;4>, <2;8>, <3;16>, or <4;32>.
- C8.** The number of processors should be function of the number of sensors: one sensor needs one processor; 2 sensors need 2 processors; 3, 4 or 5 sensors need 3 processors, more than 5 sensors need 4 processors.
- C9.** When a speed sensor is included in a configuration, then vibration feedback must be excluded and conversely.
- C10.** There should be at least one processor per kind of sensor or feedback device.

III. SPECIFYING AND ANALYZING PL WITH FINITE DOMAIN CONSTRAINT PROGRAMMING

We define a Constraint-based Language (CL) that uses CP constructs and operators in order to model product lines. The language that we propose to use is introduced in the first sub-section, then, the second sub-section illustrates its use with the working example. The last sub-section shows how to perform analysis at both the domain engineering level and the application engineering level [21].

A. The Constraint Language

Meta Model General Description. The core construct of our Constraint Language (CL) is *Constraints*, that are applied to *Variables* and *Values* and defined by *Operators* (cf. Figure 2). A variable has a domain, and a value at a certain moment. The variable domain in the CL can be boolean, integer, interval, enumeration or string.

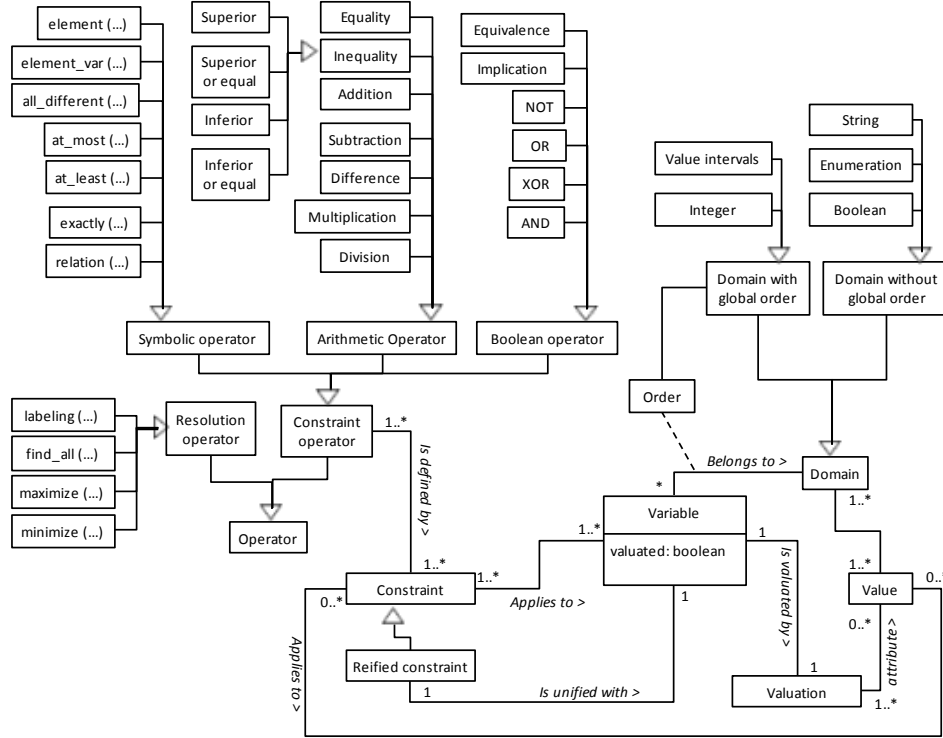


Figure 2. Constraint Language meta-model.

Operators are of two types: those for defining constraints (Constraint Operators) and those for solving constraints (Resolution Operators).

Constraint Operators are used to specify PLs. There are three types of constraint operators: Boolean, arithmetic and symbolic (which are applied on a set of variables at a time).

Resolution operators are used to solve constraints defined using the above operators to specify the PL. This allows performing various analysis operations such as verifying the PL, analysing its properties, or configuring products.

Constraints may be simple, but also reified. A reified constraint is a constraint that is unified to a variable, being itself part of another constraint. Reified constraints make it for instance possible to reason on the issue of a constraint.

Defining PL elements. Modelling PL using the CL consists in specifying constraints on PL elements (e.g., features, requirements, design fragment, components, or any other reused artefact) that are referred to using variables. Indicating that a PL element, such as a function, can be either included or excluded is simply done by giving a $[0..1]$ domain to the variables, where the 1 value would mean that the element is included in a configuration, and the 0 value that it is not.

- Specifying that elements $E1..Ek$ are boolean elements is done as follows:
`domain([E1..Ek], 0, 1)`

In the MCS example, the visual feedback is a boolean element because it can be integrated or not in a MCS. This is specified by:

`domain([Visual], 0, 1)`

Also, it may be necessary to reason on the number of times a PL element can be repeated in a product [17] (e.g., a motherboard board can contain 1, 2, 4 or more CPU cores), or to deal with elements that have enumerated or integer values (such as performance, quantity or capacity).

This can be specified with a variable E with a finite domain $[n..m]$ (m being for instance the maximum number of occurrences of E , or its maximum value).

- elements $E1..Ek$ are integer elements:
`domain([E1..Ek], n, m)`
- elements $E1..Ek$ are enumeration elements:
`domain([E1..Ek], [value_1, ..., value_n])`

In the MCS example, actuators are represented by an integer variable: `domain([Actuator], 0, 11)` and internal memory by an enumerated variable:

`domain([InternalMemory], [4, 8, 16, 32])`

The Basis of PL Modeling. Example of traditional PL elements relationships can be then specified on the $[0..1]$ domain as follows:

- Two elements $E1$ and $E2$ can only be either both present or both absent of a configuration: $E2 = E1$
- A configuration can contain an element $E2$ only if it also contains $E1$: $E2 \leq E1$
- The elements $E1$ and $E2$ cannot be simultaneously included in the same configuration: $E2 + E1 \leq 1$; for instance, in the MCS example, *Vibration* and *SpeedSensor* are mutually exclusive: $SpeedSensor + Vibration \leq 1$

- A configuration can contain a number of at least Min (or at most Max) elements within a group of $E1..Ek$ elements: $Min \leq \sum_{1..k} E_i$ and $\sum_{1..k} E_i \leq Max$
- If $E3$ is included in a configuration then either $E1$ or $E2$ is included; otherwise all are excluded: $(E3 \Rightarrow E1 + E2 = 1) \wedge (\neg E3 \Rightarrow E1 + E2 = 0)$

For instance, in the VMC example, `ConsistencyCheck` implies the inclusion of `ResponseTimeCheck`. This can be specified by the constraint:

`ConsistencyCheck \Rightarrow ResponseTimeCheck \geq 1`

- Given two sets of elements $S1 = \{E1, E2\}$ and $S2 = \{E3, E4\}$, a configuration should contain more elements from the set $S1$ than from the set $S2$: $E1 + E2 > E3 + E4$. This constraint can, of course, be extended to larger sets.
- Either $E1$ is included in a configuration, or both $E2$ and $E3$: $2 * E1 + E2 + E3 = 2$

Reasoning about Integer Variables. Two simple kinds of constraints can be specified on this type of variables:

- $E1 > a$: to indicate that element $E1$'s value shall be included at least a (i.e., it is an element that has at least a occurrences in a configuration); and
- $E1 = a$: to specify that $E1$ shall have a constant value in any configuration (i.e., the number of times an element can be included in a configuration is fixed).

More complex constraints can be specified over integer variables, for instance:

- The elements $E1$ and $E2$ are mutually excluded, this is, both can be excluded or if one is included the other can be excluded in the same configuration: $E1 * E2 = 0$
- A configuration should include more occurrences of an element than of another: $E1 > E2$
- A configuration should include as occurrences of an element $E1$ as of two other elements ($E2$ and $E3$) together: $E1 = E2 + E3$; this is for example useful to specify the $C3$ constraint of the MCS example: `Actuator = PositionSensor + DistanceSensor + SpeedSensor + Visual + Audio + Vibration.`

If in the example, n additional sensors are needed for other purposes, then the constraint would be: `Actuator + n = PositionSensor + DistanceSensor + SpeedSensor.`

- A configuration should include more occurrences of a pair of elements ($E3, E4$) than of another pair of elements ($E1, E2$) together: $E1 + E2 < E3 + E4$. This is for instance useful to specify that the number of consistency check plus the number of response time auto test sensors should be superior to

the number of memory check plus the number of functionality checks in actuators.

- The number of occurrences of $E1$ should be the half of the number of occurrences of $E2$: $2 * E1 = E2$. This can be used to specify that there should be two functionality checks auto tests per speed sensor.

Symbolic Constraints. CP over finite domains supports the specification and analysis of symbolic constraints, i.e. constraints that are checked on collections of variables. Here are some symbolic constraints:

- `alldifferent([E1, .., Ek])`: specifies that in any configuration the value of each of the $E1..Ek$ elements should be different pair wise.
- `atmost(n, [E1..Ek], a)`: specifies that at most n of the $E..Ek$ elements are equal to a .
- `atleast(n, [E1..Ek], a)`: specifies that at least n of the $E..Ek$ elements are equal to a .
- `exactly(n, [E1..Ek], a)`: specifies that exactly n of the $E..Ek$ elements are equal to a .
- `relation([E1..Ek], {[a1..ak]})`: constraints the tuple of elements $E1..Ek$ to be equal to at least one tuple in the collection of tuples $[a1..ak]$. This allows to specify extensively a predetermined collection of compatible values for $[0..n]$ elements.

In the MCS example, symbolic constraints can be used for instance to specify predefined combinations of the number of processors and the quantity of internal memory in configurations (see constraint $C7$): `relation([[1,4],[2,8],[3,16],[4,32]], [Processor, InternalMemory])`

Reified constraints. In CP, the reification of a constraint C into a variable C of the $[0..1]$ domain is achieved by a constraint:

`C \Leftrightarrow constraintC`

that establishes a correspondence between a constraint `constraintC` and C as follows: `constraintC` shall be verified in a configuration iff C is true (thus the other way round C is true iff `constraintC` is verified).

For instance, some constraints should be verified in a configuration only if some elements are included / excluded from this configuration:

- $E1 = 1 \Rightarrow C$: whenever $E1$ is included, the constraint `constraintC` reified with the C variable should be satisfied.
- $E1 = 0 \Rightarrow C$: whenever $E1$ is excluded, the constraint `constraintC` reified with C should be satisfied. Of course, these reification constraints could also be directly specified as:

$E1 = 1 \Rightarrow \text{constraintC} \wedge$
 $E1 = 0 \Rightarrow \text{constraintC}.$

In the MCS example, it would for instance be possible to generate PL models from the MCS PL model to specify sub families of MCS. One interesting such kind of sub family is this in which a position actuator is associated with each position sensor. Another aspect of this sub-family is that it can be managed only as soon as there is a central processor with a 32 MB internal memory. The constraint can be reified as follows:

$(\text{Processor}=1) \wedge (\text{InternalMemory} = 32) \Rightarrow B$

with

$B \Leftrightarrow (\text{PositionSensor} = \text{PositionActuator})$

Likewise, reified constraints can be used to specify preferences. In the MCS example, we may prefer, for cost reasons, to not use a speed sensor when we have a distance sensor. This preference can be expressed as follows:

$\text{DistanceSensor} > 0 \Rightarrow \text{minimize}(\text{SpeedSensor})$

B. Example Illustration

Developing a constraint program that specifies a product line model and resolving it is quite straightforward. For example, the MCS product line presented in section 2 can be specified, by using the rules presented in subsection A, with the following program.

```

PositionSensor ∈ {0, 3} ∧ DistanceSensor ∈
{0, 4} ∧ SpeedSensor ∈ {0, 1} ∧ Actuator ∈
{0, 11} ∧ Processor ∈ {0, 4} ∧ [Visual,
Audio, Vibration] ∈ {0, 1} ∧
InternalMemory ∈ {4, 8, 16, 32} ∧
[ActuatorFunctionalityCheck,
ActuatorMemoryCheck,
SensorFunctionalityCheck,
SensorConsistencyCheck,
SensorResponseTimeCheck] ∈ {0, 1} ∧
Sensor = PositionSensor + DistanceSensor +
SpeedSensor ∧
Sensor ≥ 1 ∧
Actuator ≥ 1 ∧
Processor ≥ 1 ∧
1 ≤ Visual + Audio + Vibration ≤ 2 ∧
Actuator = Visual + Audio + Vibration ∧
Sensor ≥ 1 ⇒ SensorFunctionalityCheck = 1 ∧
Actuator ≥ 1 ⇒ ActuatorFunctionalityCheck
= 1 ∧
SensorConsistencyCheck ⇒
SensorResponseTimeCheck ∧
relation([[1,4],[2,8],[3,16],[4,32]],
[Processor, InternalMemory]) ∧
relation([[1,1],[2,2],[3,3],[4,3],[5,3],[6,
4],[7,4],[8,4]], [Sensor, Processor]) ∧
SpeedSensor + Vibration ≤ 1 ∧
Processor ≥ SpeedSensor +

```

$\text{min}(\text{PositionSensor}, 1) + \text{min}(\text{DistanceSensor}, 1) + \text{Visual} + \text{Audio} + \text{Vibration}.$

Figure 3 illustrates the constraints set specifying the MCS example and their interdependencies. Any constraint solver can be then used to solve the program. For instance, we use GNU Prolog [13] to analyze the MCS system. For instance, we obtain a list of 1044 products that were generated in 16 milliseconds (CPU time).

Operations allowing configuration and analysis are presented in the following sub-section.

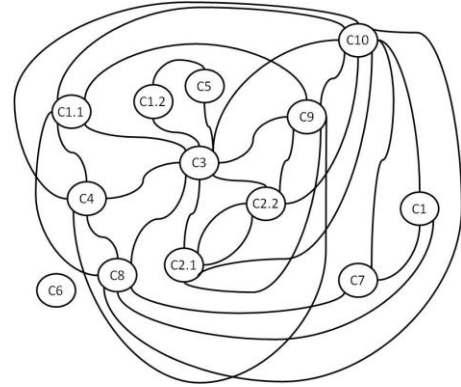


Figure 3. The MCS product line modeled as a network, where nodes represent constraints and edges indicate that concerned constraints share at least one variable

C. PL Analysis

As shown in the Figure 4, PL model and configuration requirements are expressed into constraints to allow analysis. Analysis can be done at the domain level to reason on the PL itself, as well as the application level, mainly by generating either partial or full configuration (which corresponds to a full valuation of variables).

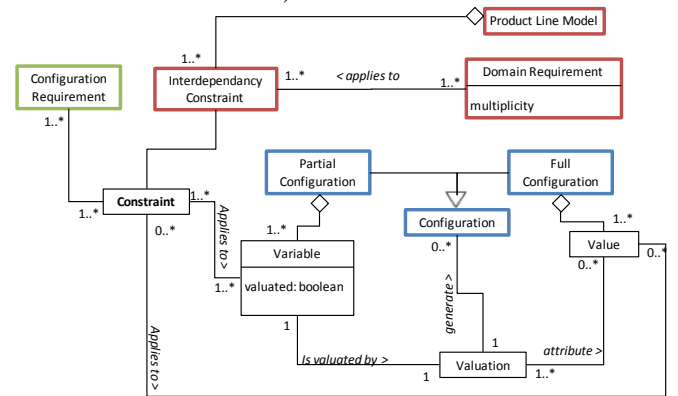


Figure 4. Product line analysis using the Constraint Language.

Domain Level Analysis. Domain analysis level is performed by specifying and resolving some constraints that the PL must respect. These “meta-constraints” can be used for verification purposes, or to optimize the PL itself.

Examples of PL verification are:

- structural verification (e.g., two different elements cannot have the same name);
- expressiveness verification criteria (e.g., the PL must not be void or with only one product);
- error-free verification (e.g., the PL must not include dead elements);
- consistency verification (e.g., the PL must not include false optional elements);
- the calculus of PL core elements, variant elements, PL homogeneity and PL commonality.

Examples of PL optimization are the maximization of reuse (e.g., any generated configuration must include at least k elements), and the minimization of components cost (e.g., the maximum cost of any generated configuration should not exceed a certain value).

Application Level Analysis. At this level, we consider a PL modeled using the constraint language. To perform analysis we make use of CL resolution operators. Analysis operations as listed in [4] can be assumed as follows (with `var` the list of the PC variables):

- One product: `labeling(var)` allows to calculate one valid solution for the modeled problem. Then, alternative solutions can be calculated successively.
- All products: `find_all(var, fd_labeling(var),L)` allows enumerating all problem solutions in the list `L`.
- Number of products: this operation can be performed while counting the result list length of the `find_all()` operator, which literally gives `find_all((X,Y), labeling([X,Y]),L), length(L,N)`. `N` is the number of possible solutions.
- Optimization: `maximize(labeling(Vars),X)` allows to find a solution such that the objective function `X` is maximized. Conversely, `minimize(labeling(Vars),X)` allows to find a solution that minimize the objective function `X`. This function can be a constraint on for example the cost or the benefit of PL devices. Optimization operation can be used also domain level constraints (see above).
- Valid product: a valid product is an assignment for all the variables of the PC problem that does not violate its constraints. Then, a product is valid if his resolution, together with the PL constraints, does not arise errors.
- Valid partial configuration: likewise, a partial configuration is an assignment for some PC problem variables that does not violate its constraints. Then, a partial configuration is valid if his resolution, together with the PL constraints, gives a valid total configuration.
- Products that contain a given set of features: pre-selecting some PL features means, in terms of PC, that a TRUE value is assigned to the corresponding

variables. Calculating products that contains a given set of features boils down to resolve the PC problem in which a TRUE value is assigned to the desired features.

- Multistep configuration: as discussed above, decisions over steps can be programmed using reified constraints.
- Dependency analysis having a partial configuration: this operation returns a new configuration with the features that should be selected and removed as a result the propagation of the PL constraints. In terms of CP, dependency analysis is performed by resolving the PL problem augmented with the assignment of the partial configuration.

IV. REPRESENTATION OF PRODUCT LINE MODELS AS CONSTRAINT PROGRAMS

CP is a powerful paradigm that can deal with numerous variability notations. Table 1 illustrates transformations patterns of some common constraints to all PLM representations. For reasons of succinctness, others specific constraints are not introduced in this paper.

Another use of constraints on the $[0..n]$ domain in PL is to apply them to element attributes, as proposed in [18] with attributes associated to $[0..1]$ elements. In [19], we have demonstrated how to specify constraints on attributes to reason on goal based product configuration, to guide for example a cost/benefit analysis of products during their configuration.

Besides, reified constraints can be used in staged configuration. In [20], Czarnecki et al. define staged configuration through specialization steps. A specialization takes a PL model as input, and produces another PL model, where certain configuration choices are eliminated. The fundamental of staged configuration is then to enable extra constraints that shall be associated to a configuration model, which shall itself be considered as a PL model. Staged configuration can be found useful when not all constraints shall be verified at once, but enabled in an ordered-fashion. Reified constraints allow defining such constraints that are enabled when some conditions are satisfied in a given step.

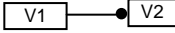


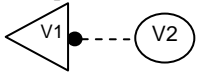
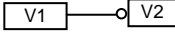
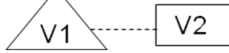
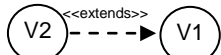
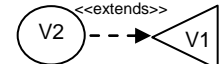
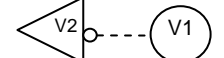
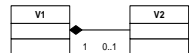
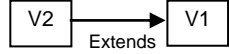
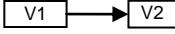
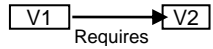

The constraints of a PL model that shall only be verified at a stage of configuration identified must be reified. Identifying stages of configuration can be done either using FD variable that represents time (the version number of the configuration) [7], or it can be conditioned by the inclusion / exclusion of some elements in the configuration.

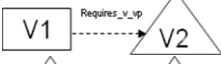
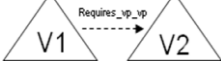
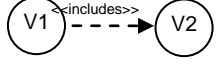
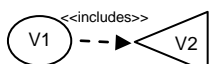
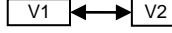
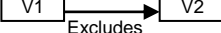
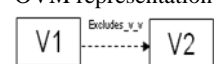
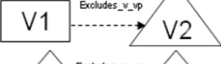
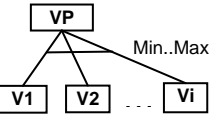
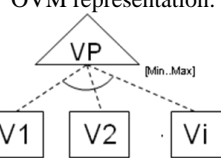
Reified constraint can also be used to specify constraint over decision points, as in [21] as follows. Assuming that a decision point `D` is specified using constrains `C1..Cn`. A constraint `C` on `D` shall simply be expressed as:

$$C \Rightarrow D, \text{ where } D \Leftrightarrow C1 \wedge \dots \wedge Cn$$

to indicate that whenever condition `C` is met (e.g., an element is included in a configuration), the constraints associated with decision point `D` shall be satisfied.

TABLE I. PATTERNS FOR TRANSFORMING PL MODELS, REPRESENTED BY MEANS OF DIFFERENT FORMALISMS, INTO CP

Names and Semantics	Representations	CP Representations
Mandatory If the father element ($V1$) is selected, the child element ($V2$) must be selected as well and vice versa.	FODA representation  Class representation  OVM representation  Use case representation 	$V1 > 0 \Leftrightarrow V2 > 0$ (i.e., if $V1 > 0$ then $V2 > 0$ and vice versa)
Optional If the father element ($V1$) is selected, the child element ($V2$) can but needs not be selected. Otherwise, if $V2$ is selected, the father element ($V1$) must as well be selected.	FODA representation  OVM representation  Use Case representation [32]    Class representation [30,31]  Ontology representation [33] 	$V2 \leq V1$ (i.e., if $V1 = 1$ then $V2 = 1$ and if $V2 = 1$ then $V1 = 1$)
Requires If element $V1$ is selected, the required element $V2$ has to be selected as well, but not vice-versa, that is, $V2$ can be selected when $V1$ is not	Feature representation  Ontology representation  OVM representation 	$V1 > 0 \Rightarrow V2 > 0$ (i.e., if $V1 = 1$ then $V2 \geq 1$; if $V1 = 0$ then $V2$ is not restricted by $V1$)

selected.   Use Case representation  		
Exclusion Indicates that both elements ($V1, V2$) cannot be selected in one product configuration.	Feature representation  Ontology representation  OVM representation  	$V1 * V2 = 0$ Or if $V1$ and $V2$ are Boolean variables: $V1 + V2 \leq 1$ (if $V1 > 0$ then $V2 = 0$; if $V2 > 0$ then $V1 = 0$ and if $V1 = 0$ then $V2 \geq 0$ and vice versa)
Group cardinality Cardinality determines how many variants (with the same father) may be chosen, at least <i>Min</i> and at most <i>Max</i> of the group. Besides, if one of the children is selected, the father element must be selected as well.	Feature representation:  OVM representation: 	$V1 \Rightarrow VP \wedge$ $V2 \Rightarrow VP \wedge \dots$ $V_i \Rightarrow VP \wedge$ $VP \geq 1 \Rightarrow$ $V1 + V2 + \dots + V_i$ $\geq \text{Min} \wedge$ $VP \geq 1 \Rightarrow$ $V1 + V2 + \dots + V_i$ $\leq \text{Max}$ i.e., (Min=0, Max=2 and PV=5) then zero, one or two elements among $\{V1, V2, \dots, V_i\}$ can be chosen.

Note that (a) we do not deal with individual cardinalities and (b) when we talk about cardinality-based feature models we are considering only group cardinalities. Cloned features [17] and multi-valued variables are partially treated by Karataş et al. [28]. So, questions like: what is the semantics of cross-tree constraints in presence of feature cardinalities? What does it mean that A requires B, if B has a parent with more than singleton cardinality? Is the semantics existential? Is it universal? Anything in between? What does $F1.A = F2.B + F3.C$ if any of the involved features is in a cloned sub-tree? Are open issues and are proposed as future work.

V. EVALUATION

We evaluate the effectiveness of our approach by testing its implementation and application feasibility.

Tool implementation. We developed an interactive environment composed of two tools: VariaMos [26] and GNU Prolog [13]. VariaMos is an Eclipse plug-in that allows constraint programs management (e.g., creating, editing and saving), verification (detect void models, false product line models, dead variables, false optional variables, not attainable domains and redundant constraints), analysis (supporting all analysis functions documented by Benavides et al. in their latest literature review [25], see section 3.3 of this paper) and specification of requirements (e.g., configure a product, define a filter or a partial configuration and specify extra constraints or particular requirements). A snapshot screen of our tool VariaMos’s interface is shown in Figure 5. GNU Prolog is our CP solver.

Feasibility study with a real case application. One particular question that can be raised about the new kinds of constraints that have been identified in this paper is “are they useful?” Although only long term experience shall provide a definitive answer to this question, one might be interested in looking for special constraints that could be specified in a real case. To do so, we have used our CP over Finite Domains (FD) approach to specify constraints on a family of blood analysis automatons [19] in the context of a cooperation with the partner industry STAGO.

Using FD constraints allowed us to specify the same constraints as the one that we had identified to reason about cost and revenue of each feature. To do so, we associated $[0..n]$ attributes to each features to specify costs and benefits. We had to define a fix value for n – we chose to use the same maximum cost and revenue for all features for the purpose of the study. For example, we specified constraints on the minimal number of measurement wells depending on the required tests and the required cadence for these tests.

```
Chronometric.NumberOfWells +
Colorimetric.NumberOfWells +
Immunologic.NumberOfWells ≥
max(LaunchTest.TestCadence) *
max(LaunchTest.TestDuration)
```

We could also specify that the initially optional function ‘Agitate’ must be implemented whenever one of the tests TCA, ATIII or PC are not included.

```
(LaunchTest.TestType ≠ TCA) ∨
(LaunchTest.TestType ≠ ATIII) ∨
(LaunchTest.TestType ≠ PC) ⇒ Agitate = 1
```

Looking at our list of specific constraints, we identified the following constraints which could not be specified with $\{true, false\}$ features, but can be specified with our integer constraint notation:

- constraints on both $[0..n]$ features and feature attributes. For example, we could play with the number of chronometric, colorimetric and immunologic measures and specify a constraint on the number of their occurrence with regard to the cadence and duration of the test.

```
Chronometric + Colorimetric + Immunologic
≥ LaunchTest.TestCadence *
LaunchTest.TestDuration
```

- symbolic constraints such as:
 $Atmost(1, [Agitate, Mix, Incubate], 2)$ to specify that each activity in a methodology can be repeated at most twice;
- another example of use of symbolic constraints was to specify possible combinations of value of the cadence, duration, and kind of determination for different kinds of test types:
 $Relation([LaunchTest.TestType, LaunchTest.TestDuration, LaunchTest.TestCadence, determination], [[TP, 2, 14, simple], [TP, 2, 14, double], [TCA, 2, 14, simple], [TT, 3, 2, double], [Fib, 10, 5, double], [ATIII, 15, 3, double], [VwF, 13, 8, double], [PC, 2, 6, simple], [DDi, 6, 8, simple]])$
- Last, we were able to specify reified constraints such as:
 $LaunchTest.TestType = TCA \Leftrightarrow C$
 $C \Rightarrow Chronometric = 1 \wedge Chronometric.Speed = normal$
 which enforces the use chronometric measurement technique when TCA test is demanded. It specifies also the required speed for this test.

We also used feature attributes to support cost/benefit analysis on measurement techniques. The following goals could for instance be specified:

```
Min (Chronometric.Cost *
Chronometric.NumberOfWells +
Colorimetric.Cost *
Colorimetric.NumberOfWells +
Immunologic.Cost *
Immunologic.NumberOfWells)
∧ Max (Chronometric.Revenue *
Chronometric.NumberOfWells +
Colorimetric.Revenue *
Colorimetric.NumberOfWells +
Immunologic.Revenue *
Immunologic.NumberOfWells)
```

Figure 5 shows the STAGO product line model specified as a constraint program and the configuration interface of our tool VariaMos. Note that some constraints like the symbolic constraint $(1, [Agitate, Mix, Incubate], 2)$ discussed before, can be specified not as a constraint of the product line but as a particular constraint on configuration stage. Note also that the value of each feature can be specified in the configuration interface for a particular product (assigning a value to all features) or for a collection of products (assign a value to some features) where the assignation of the value 0 to a feature means that the feature must not belongs to configured product(s).

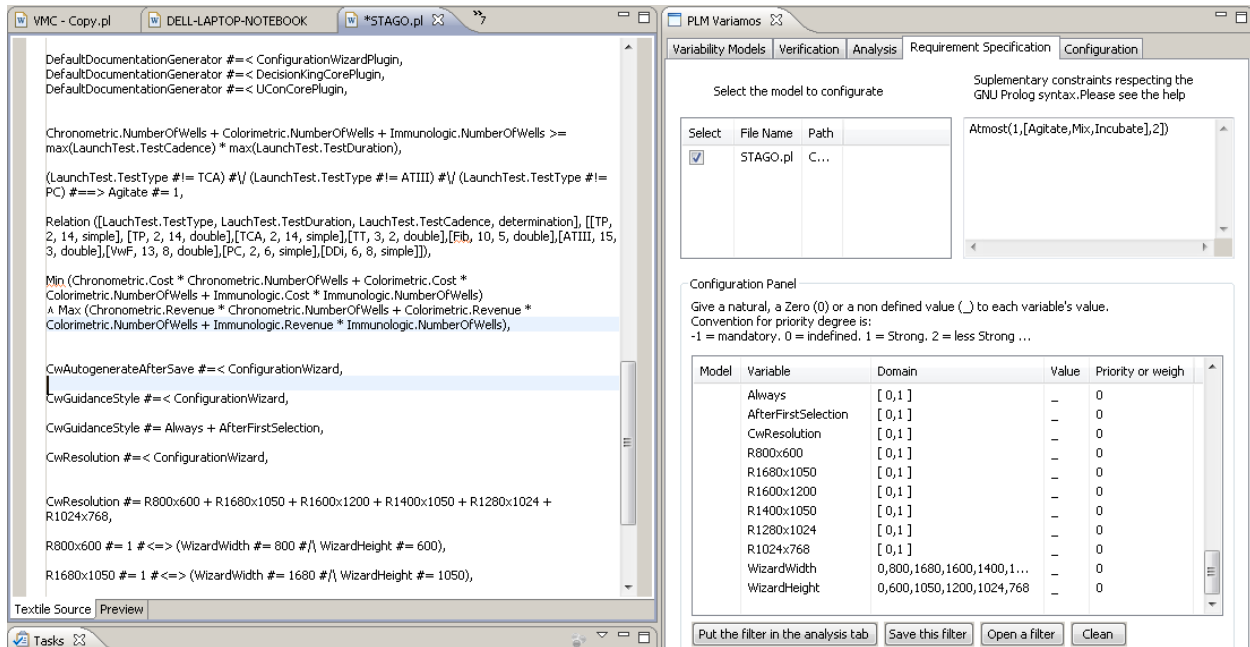


Figure 5. Snapshot of our tool VariaMos. STAGO product line model representation as a constraint program and its configuration interface.

Our observations are also the following ones:

- Incremental development and maintenance of PL models is made possible as long as models are modified by adding/removing constraints.
- We use GNU-Prolog as solver. It computes very efficiently a first complete solution w.r.t. the selected/excluded features. In practice, this helped us in the configuration process as it provided a general idea of the product that was being built.
- GNU-Prolog computation of the next solution was effective as it offered an alternative to the configurations that had already been explored. Iterating over this function allowed to review the various solutions one by one – or to identify that the variability space was still very open by counting the number of remaining configurations that satisfied the constraints for the requirements at hand.

These results are encouraging and confirm that CP over FD is well suited to precisely model and efficiently configure PL.

VI. RELATED WORKS

This paper is not the first to explore the use of constraint programming in the context of PL. Some proposals had been made to support automatic analysis of feature-based models in order to allow retrieving information. The largest number of works to automate features analysis is based on propositional logic [1, 2, 3, 18].

Batory was the first to use SAT solvers to analyze feature models. In these constraints, features are Boolean variables (either they are included or not in a configuration).

Czarnecki's proposals of staged configuration, features cardinalities and feature attributes have created an opportunity to move from Boolean to integer constraints

specification. Our approach belongs to this family of approaches that relies on integer domain constraints rather than on Boolean ones. The simple fact of replacing the $\{true, false\}$ domain by $[0..1]$ opens the door to kinds of constraints that did not exist in the aforementioned approaches.

In particular, Benavides's works [4, 18] have shown how feature models could be analyzed by specifying integer constraints on attributes associated with features. In Benavides's approach, features themselves still have a $\{true, false\}$ domain, while our approach allows dealing with $[0..n]$ features.

White et al. [7] also provide a CP support for multi-step configuration over time, while respecting resource constraints. We believe reification constraints able to deal with progressive configuration either by providing successive complete products as in [7] or successive partial configurations as in [20].

Besides, aforementioned approaches consider only single monolithic feature models. As shown in section 4, our approach provides the ability to deal with multiple models that are specified using different variability languages.

Furthermore, our approach explores more FD Constraint Programming capabilities that have not been exploited so far. It provides numerous types of constraints (e.g. symbolic and reified constraints) that have not been proposed by any of the approaches referenced before.

VII. CONCLUSION

CP has proved a powerful paradigm for solving combinatorial problems arising in many domains, such as scheduling, planning, vehicle routing, configuration, networks or bio-informatics. Our proposal is to specify product lines as a finite domain constraint program i.e. not

just a boolean program that implements features selection in a FODA-like models, but a series of integer constraints that apply to anything that varies and has configuration constraints.

We believe our approach is original as (a) it is a first attempt to integrate various variability models through a unique representation, (b) it supports direct reasoning on product line models (c) it supports the specification of complex configuration requirements.

Nonetheless, some further work is required for the multi-valuated PL elements, on which constraints may need some adjustments. Besides, the approach can be extended to deal with reals, which can for example allow performing some probabilistic reasoning (some industries like Renault have expressed the need to plan pieces logistics). We have explored constraint programming on finite domains, but many other domains could be relevant: Intervals, Trees, Lists, and Sets. Constraint Programming is versatile in that it adapts quite well to different applications. We have little doubt that the systematic exploration of these domains will generate new knowledge about product lines engineering.

REFERENCES

- [1] Mannion M.: Using first-order logic for product line model validation. In Proceedings of the Second SPLC, LNCS 2379, pages 176–187, San Diego, CA, Springer (2002)
- [2] Zhang W., Zhao H., and Mei H.: A propositional logic-based method for verification of feature models. In J. Davies, editor, ICFEM 2004, volume 3308, pages 115–130. Springer–Verlag (2004)
- [3] Batory D.: Feature models, grammars, and propositional formulas. In Software Product Lines Conference, volume 3714 of Lecture Notes in Computer Sciences, pages 7–c. Springer–Verlag (2005)
- [4] Benavides D., Ruiz-Cortés A., and Trinidad P.: Using constraint programming to reason on feature models. In The Seventeenth International Conference on Software Engineering and Knowledge Engineering, SEKE 2005, pages 677–682 (2005)
- [5] White J., Schmidt D., Benavides D., Trinidad P., and Ruiz-Cortés A.: Automated diagnosis of product-line configuration errors in feature models. In Proceedings of the SPLC (2008)
- [6] Trinidad P., Benavides D., Duran A., Ruiz-Cortés A., and Toro M.: Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6):883–896 (2008)
- [7] White J., Dougherty B., Schmidt D., and Benavides D.: Automated reasoning for multi-step software product line configuration problems. In Proceedings of the SPLC, pages 11–20 (2009)
- [8] Salinesi C., Mazo R., Diaz D. Criteria for the verification of feature models, In 28th INFORSID Conference, Marseille, France (2010)
- [9] Jaffar J. and Maher M. J.: Constraint logic programming: A survey. *Journal of Logic Programming*, Vol. 19/20 (1994)
- [10] Le Berre D. SAT4J solver, www.sat4j.org. Last consulted 01/03/2010
- [11] Sheldon B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6):509–516 (1978)
- [12] SMV system, www.cs.cmu.edu/~modelcheck. Last consulted 01/03/2010
- [13] Diaz D. and Codognet P. Design and Implementation of the GNU Prolog System. *Journal of Functional and Logic Programming (JFLP)*, Vol. 2001, No. 6 (2001). Available to download from: <http://www.gprolog.org/>
- [14] CHOCO solver, <http://www.emn.fr/z-info/choco-solver/index.html>.
- [15] Jaffar J., Michaylov S., Stuckey P. and Yap R. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3), (1992)
- [16] Colmerauer A.: An Introduction to Prolog III. *Communications of the ACM*, vol. 33, no. 7, (1990)
- [17] Czarnecki K., Helsen S., and Eisenecker U.W.: Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29 (2005)
- [18] Benavides D.: On the Automated Analysis of Software Product Lines Using Feature Models. A Framework for Developing Automated Tool Support. University of Seville, Spain, PhD Dissertation (2007)
- [19] Djebbi O., Salinesi C.: RED-PL, a Method for Deriving Product Requirements from a Product Line Requirements Model. *International Conference on Advances in Information Systems Engineering, CAISE'07*, Norway (2007)
- [20] Czarnecki K., Helsen S., and Eisenecker U.: Staged configuration through specialization and multi-level configuration of feature models. *Software Process Improvement and Practice*, 10(2) (2005)
- [21] Pohl K., Bockle, G., van der Linden, F.J. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer (2005)
- [22] Mazo R., Grünbacher P., Heider W., Rabiser R., Salinesi C., Diaz D. Using Constraint Programming to Verify DOPLER Variability Models. *Proceedings of the VaMos Workshop, Namur-Belgium, January 27th-29th*, ACM Press (2011)
- [23] Salinesi C., Rolland C., Mazo R.: VMWare: Tool Support for Automatic Verification of Structural and Semantic Correctness in Product Line Models. In *VaMos Workshop, Spain* (2009)
- [24] Djebbi O., Salinesi C., Diaz D.: Deriving Product Line Requirements: the RED-PL Guidance Approach. *Asian Pacific Software Eng. Conference (APSEC)*, Japan (2007)
- [25] Benavides D., Segura S., Ruiz-Cortés A. *Automated Analysis of Feature Models 20 Years Later: A Literature Review*. Information Systems . Elsevier (2010)
- [26] Mazo R. *VariaMos tool*, available to download at <https://sites.google.com/site/raulmazo/>
- [27] Kang K., Cohen S., Hess J., Novak W., Peterson S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
- [28] Serkan Karataş A., Oğuztüzün H., Dođru A. Mapping Extended Feature Models to Constraint Logic Programming over Finite Domains. *SPLC, Korea* (2010)
- [29] Salinesi C., Mazo R., Diaz D., Djebbi O. Solving Integer Constraint in Reuse Based Requirements Engineering. *18th IEEE International Conference on Requirements Engineering*. Sydney, Australia (2010)
- [30] Korherr B., List B. A UML 2 Profile for Variability Models and their Dependency to Business Processes. *1st International Workshop on Enterprise Information Systems Engineering (WEISE 07)*, September 2007, Regensburg, Germany, IEEE Press (2007)
- [31] T. Ziadi. *Manipulation de Lignes de Produits en UML*. PhD thesis, IRISA-TRISKELL, Université de Rennes 1 (2004)
- [32] van der Maßen T., Lichter H. Modeling Variability by UML Use Case Diagrams. *International Workshop on Requirements Engineering for Product Lines (REPL'02)*, pages 19–25 (2002)
- [33] Abo Zaid L., Houben G-J., De Troyer O., Kleinermann F. An OWL-Based Approach for Integration in Collaborative Feature Modelling. *Workshop on Semantic Web Enabled Software Engineering*, (2008)