Constraints: the Heart of Domain and Application Engineering in the Product Lines Engineering Strategy

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

International Journal of Information System Modeling and Design IJISMD. ISSN 1947-8186, eISSN 1947-819. April-June 2012, Vol. 3, No. 2.

Constraints: the Heart of Domain and Application Engineering in the Product Lines Engineering Strategy

Raúl Mazo, University of Antioquia & CRI Panthéon Sorbonne University, France
Camille Salinesi, CRI Panthéon Sorbonne University, France
Daniel Diaz, CRI Panthéon Sorbonne University, France
Olfa Djebbi, CRI Panthéon Sorbonne University, France
Alberto Lora-Michiels, Baxter International Inc, Belgium

ABSTRACT

Drawing from an analogy between features based Product Line (PL) models and Constraint Programming (CP), this paper explores the use of CP in the Domain Engineering and Application Engineering activities that are put in motion in a Product Line Engineering strategy. The start idea is simple: both CP and PL engineering deal with variables, and constraints that these variables must satisfy. Therefore, specifying a PL as a constraint program instead of a feature model, or another kind of PL formalism, carries out two important qualities of CP: expressiveness and 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 most PL languages such as the Feature-Oriented Domain Analysis (FODA). Specifying boolean, arithmetic, symbolic and reified constraint, provides a power of expression that spans beyond that provided by the boolean dependencies in FODA models. On the other hand, PL models expressed as constraint programs can directly be executed and analyzed by off-the-shelf solvers. Starting with a working example, this paper explores the issues of (a) how to specify a PL model using CP, including in the presence of multi-model representation, (b) how to verify PL specifications, (c) how to specify configuration requirements and (d) how to support the product configuration activity. Tests performed on a benchmark of 50 PL models show that the approach is efficient and scales up easily to very large and complex PL specifications.

Keywords: Computer science, information systems, constraints, product line specification, product line reasoning, product line analysis, product line verification, product line integration, product line configuration, constraint-based product lines, transformation of product lines into constraint programs.

INTRODUCTION

Many experiences in the industry have shown that Product Lines engineering is an effective way to deal efficiently with reuse during analysis, design, development, test or even delivery of series of products that contain similar and varying features. Starting from 3 products, the Product Line engineering strategy has positive effects on time to market, product quality and customer satisfaction (Clements and Northrop 2001). Product Line success stories gathered in the SPLC

"Hall of Fame" 1 show companies such as Boeing, Cummins, HP, Nokia, Philips Medical Systems, or Toshiba benefited from the Product Line strategy in many ways, spanning from a drastic reduction of time to market, number of defects per product, engineering effort to deploy and maintain products, combined with a substantial increase in the number of products that can be deployed. The business benefits are multiple: reduced time to revenue, higher profit margins, improved ability to aim at market windows, higher profit margins, reduced risk in product deployment, and even improved reputation of the company due to better product quality2.

The Product Line engineering strategy entails two activities: domain engineering and application engineering.

Application engineering consists in analysing, designing, building, customizing, or testing one product *by reuse*. Different kinds of artefacts can be reused: requirements, design fragments, architecture, code, test cases, etc. The reuse strategy is based on the exploitation of Product Line models built during domain engineering.

Domain engineering consists in specifying artefacts *for reuse*. This means specifying the artefacts to make them readily reusable, as well as specifying their reuse conditions. Many different specification languages have been proposed to support this. The most well known is probably FODA and its dialects (Kang *et al.* 1990). However OVM (Pohl *et al.* 2005), UML extensions (Ziadi 2004), (van der Maßen, Lichter 2002), Dopler (Dhungana *et al.* 2010), the text-based variability language (TVL, cf. Boucher *et al.* 2010 and Classen *et al.* 2011) and the DSL proposed in (Mannion 2002) are noteworthy alternatives as they allow specifying Product Lines with complementary viewpoints such as marketing, architecture, logistics, maintenance, etc.

In our view, the approaches that exploit these formalisms have in common that they emphasize the role of constraints at both the level of domain engineering and application engineering. Indeed, domain engineering can be seen in these approaches as the specification of variables ("features", "attributes", "variation points", etc) and constraints ("dependencies"). Application engineering then consists in defining values for these variables, while ensuring that the constraints are satisfied. Therefore, variables specify what can vary from a product to the other, in other terms the characteristics of artefacts that can be reused. On the other hand, constraints specify the reuse conditions, *ie* when artefacts can (or should) be reused or not.

Our approach is grounded on former research works that proposed to transform traditional Product Line Models (PLMs) into propositional logic (Mannion 2002, Zhang *et al.* 2004, Batory 2005, White *et al.* 2008) or boolean constraints programs in order to reason about them (Benavides *et al.* 2005, Trinidad *et al.* 2008, Mendonça *et al.* 2009). As these works showed it, using traditional Product Line specification formalisms raises a series of problems:

- being different they are necessary to specify multiple views, but at the same time they are difficult to integrate;
- they are often quite limited in the sense that they are not rich enough to specify complex requirements or configuration constraints;
- they are contemplative, and therefore it is difficult to automate verification, analysis and configuration activities.

The approach taken in this paper makes a step beyond the aforementioned ones: we believe that constraints programming should be at the core of product line engineering rather than just a

[1] http://www.splc.net/fame.html

[2] http://www.softwareproductlines.com/benefits/benefits.html

tool or a by-product issued after transforming classical models. Not only this allows specifying domain models that would not be specified when starting with traditional formalisms, but also it allows integrating multiple views, it provides users with a rich language to specify their requirements, and it brings Product Line specifications that are readily available for automated reasoning using constraints solvers.

Different kinds of constraint programs can be built; each can be solved with a specific kind of solver. The choice depends on the types of variables on which the reasoning applies (Jaffar and Maher 1994). :

- Boolean variables can be treated with SAT (Le Berre 2010), BDD (Akers 1978), or SMV (Specification and Verification Center 2010);
- Integer can be handled with GNU Prolog (Diaz and Codognet 2001), or CHOCO (Ecole de Mines de Nantes 2010);
- Reals can be handled for instance using clp(R) (Jaffar *et al.* 1992) ;
- Trees and Lists can be handled with Prolog-III (Colmerauer 1990).

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 (Benavides *et al.* 2005), (White *et al.* 2009). 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.

Only few approaches have dealt with integer CP (or finite domain CP) (Benavides *et al.* 2006, Djebbi *et al.* 2007, Salinesi *et al.* 2010a and 2010b, Mazo *et al.* 2011a). In our previous works (Mazo *et al.* 2011d, Salinesi *et al.* 2011), we observed that transforming feature models into integer CP provides many advantages
FODA dialects (such as attributes or feature cardinalities), specifying more complex requirements than select/de-select a feature, or making complex analyses and verifications (Salinesi *et al.* 2010a and 2010b, Mazo *et al.* 2011a).
Another observation is that most existing approaches consist in transforming existing PL models into CP. We believe that this way of working hinders the full exploitation of the versatility of CP. Our research goal is to *explore the expressiveness of constraint programming to specify product line models and to support its automation and reasoning*. This goal has two facets: (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 and configuration 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 real PL (Djebbi and Salinesi 2007). This allowed us both to evaluate the feasibility of the approach, but also to explore the analysis capability offered by off the shelf constraint solvers supporting the chosen integer CP language. The approach was then discussed with PL experts of companies like ADN, Renault, Stago and Baxter (Salinesi *et al.* 2010). Besides, we developed a series of transformation strategies to specify FODA models (Kang et al. 1990), UML-based variability models (Ziadi 2004), (van der Maßen, Lichter 2002) and OVMs (Pohl *et al.* 2005) using integer CP (Djebbi *et al.* 2007). Then, we explored different case studies to evaluate our approach and further develop it (Mazo *et al.* 2011d), (Salinesi *et al.* 2011). Last, we experimented the performance and scalability of our approach using a large benchmark (Mazo *et al.* 2011d).

One driving 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, even though OCL could be used to specify PL constraints. The reason for this decision was that even if OCL is a well known language to represent constraints, OCL rules are executed by an interpreter and not by a solver, loosing, in the way, some reasoning capabilities important in the domain of product lines, for instance to calculate the number of valid products represented in the product line model.

The rest of the paper is structured as follows: Section II introduces the UNIX working example, which is used in the rest of the paper to illustrate our approach. Section III describes the approach by presenting CP with a meta-model and the various kinds of constraints that can be used to specify a PL. In addition, Section III presents the mapping between several product line modelling languages and our constraint programming approach to represent product line models. Once product line models are represented as constraint programs, we show how to integrate them in Section IV, verify them in Section V, analyze them in Section VI, and configure products from them in Section VII. Section VIII presents the evaluation of our approach from the implementation, computation scalability, feasibility with real cases and usability points of view. Section IX discusses works related to the specification of PLMs by means of CP. Other works that deal with other CP aspects are spread out throughout the paper. To finish, Section X concludes the paper by summarizing its outcomes and presenting some research directions.

WORKING EXAMPLE

The example taken in this paper is that of the UNIX operating system. UNIX was first developed in the 1960s, and has been under constant development ever since. As other operating systems, it is a suite of programs that makes computers work. In particular, UNIX is a stable, multi-user and multi-tasking system for many different types of computing devices such as servers, desktops, laptops, down to embedded calculators, routers, or even mobile phones. There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are Sun Solaris, Berkeley (BSD), GNU/Linux, and MacOS X.

The UNIX operating system is made up of three parts: the kernel, the shell and the programs; and two constituent elements: files and processes. Thus, these three parts consist in a collection of files and processes allowing interaction among the parts. The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the file-store and communications in response to system calls. The shell acts as an interface between the user and the kernel, interprets the commands (programs) typed in by users and arranges for them to be carried out. As an illustration of the way the shell, the programs and the kernel work together, suppose a user types *rm myfile* (which has the effect of removing the file *myfile*). The shell searches the file-store for the file containing the program *rm*, and then requests the kernel, through system calls, to execute the program *rm* on *myfile*. The process *rm* removes *myfile* using a specific system-call. When the process *rm myfile* has finished running, the shell gives the user the possibility to execute further commands.

As for any Product Line, our example emphasizes the common and variable elements of the UNIX family and the constraints among these elements. This example is built from our experience with UNIX operating systems and it does not pretend to be exhaustive, neither on the constituent elements nor on the constraints among these elements. The example is presented with two views. The first view is about the technical aspects of UNIX; for instance, the technical specification of the screen resolution according to the available types of interface. To depict this

view, we propose eight constraints and their corresponding representation in CP. The second view is the one of final users; for instance, it looks at what utility programs or what kinds of interfaces are available for a particular user.

Technical view:

Constraint 1. UNIX has one KERNEL.

Constraint 2. Some mandatory functions of the KERNEL are:

- ALLOCATING THE MACHINE'S MEMORY to each PROCESS
- SCHEDULING the PROCESSES
- ACCOMPLISHING THE TRANSFER OF DATA from one part of the machine to another

Constraint 3. UNIX has zero or several PROCESSES for each user. For the sake of simplicity will consider only two users in this running example: ROOT_USER and GUEST_USER. The collection of PROCESSES varies even when the UNIX product is fully configured.

Constraint 4. UNIX offers a logical view of the FILE SYSTEM. A FILE SYSTEM is a logical method for organising and storing large amounts of information in a way that makes its management easy.

Constraint 5. The KERNEL is composed of static or dynamic software modules. If the kernel was compiled for a specific hardware platform and cannot be changed, it is called a static Kernel. If the Kernel has the ability to dynamically load modules so that it can 'adapt' to a platform, it is called a dynamic Kernel. For instance, the modules SUPPORT_USB, CDROM_ATECH, and PCMCIA_SUPPORT cannot be charged, be charged in a static way or be charged in a dynamic way. For each module, let us number these three different options 0, 1 and 2, respectively.

Constraint 6. The SHELL is a command interpreter; it takes each command and passes it to the KERNEL to be acted upon.

Constraint 7. The GRAPHICAL interface is characterized by a WIDTH RESOLUTION and a HEIGHT RESOLUTION that can have the following couples of values [800,600], [1024,768] and [1366,768].

User view:

Constraint 8. UNIX can be installed or not and the installation can be from a CDROM, a USB device or from the NET.

Constraint 9. UNIX provides several hundred UTILITY PROGRAMS for each user. The collection of UTILITY PROGRAMS varies even when the UNIX product is full-configured.

Constraint 10. The SHELL is a kind of UTILITY PROGRAM. Different USERS may use different SHELLS. Initially, the USER administrator supplies a default shell, that can be overridden or changed by users. Some common SHELLS are:

- Bourne shell (SH)
- TC Shell (TCSH)
- Bourne Again Shell (BASH)

Constraint 11. Some functions accomplished by the UTILITY PROGRAMS are:

- EDITING (mandatory and requires USER INTERFACE)
- FILE MAINTENANCE (mandatory and requires USER INTERFACE)
- PROGRAMMING SUPPORT (optional and requires USER INTERFACE)
- ONLINE INFO (optional and requires USER INTERFACE)

The USER INTERFACE can be GRAPHICAL and/or TEXTUAL.

SPECIFYING AND ANALYZING PL WITH FINITE DOMAIN CONSTRAINT PROGRAMMING

Our theory is that a Constraint Language (CL) can be used as a primary concept to model product lines. The language that we propose is introduced in the first sub-section, then, the second sub-section illustrates its use with the working example. The last sub-section shows mappings between traditional PL formalisms and our CP-based formalism for CP specification.

The Constraint Language

The core constructs, of our Constraint Language (CL) are *Constraints* and *Operators* that are applied to *Variables* and *Values*. Figure 1 presents our metamodel using UML notation (UML was chosen for the sake of clarity; an example of formal grammar of one popular CP notation can be found in (S. de Boer and Palamidessi 1991)). As the metamodel shows it, a variable has a domain, and at a given moment in time, a value. The domain of variables can be boolean, integer, interval, enumeration or string. This metamodel improves the version originally presented in (Salinesi et al. 2011) in two respects:

(i) it distinguishes between constraints and operators. Thus, a constrains can be symbolic, arithmetic or boolean and contain zero or several operators. Operators are of three types: multiplication (*), addition (+) and subtraction (-), which were considered exclusively as arithmetic operations in the previous version of the metamodel. Still, these operations can take place both in symbolic constrains (e.g., `exactly(A+B, 5)`) and in boolean constraints (e.g. `A*B` $\Rightarrow$ `C`).

(ii) Resolution operators, are no longer considered as part of the CP language to specify product line models, but to help in the automatic reasoning of product line models.

Constraints are used to specify PLs. There are three types of constraints: boolean, arithmetic and symbolic. Symbolic constraints are applied on a set of variables at a time.

Constraints may be simple, but also *reified*. A reified constraint is a constraint whose truth value can be captured with a boolean variable, which can itself be part of another constraint. Reified constraints make it for instance possible to reason on the realisation of constraints at different times.

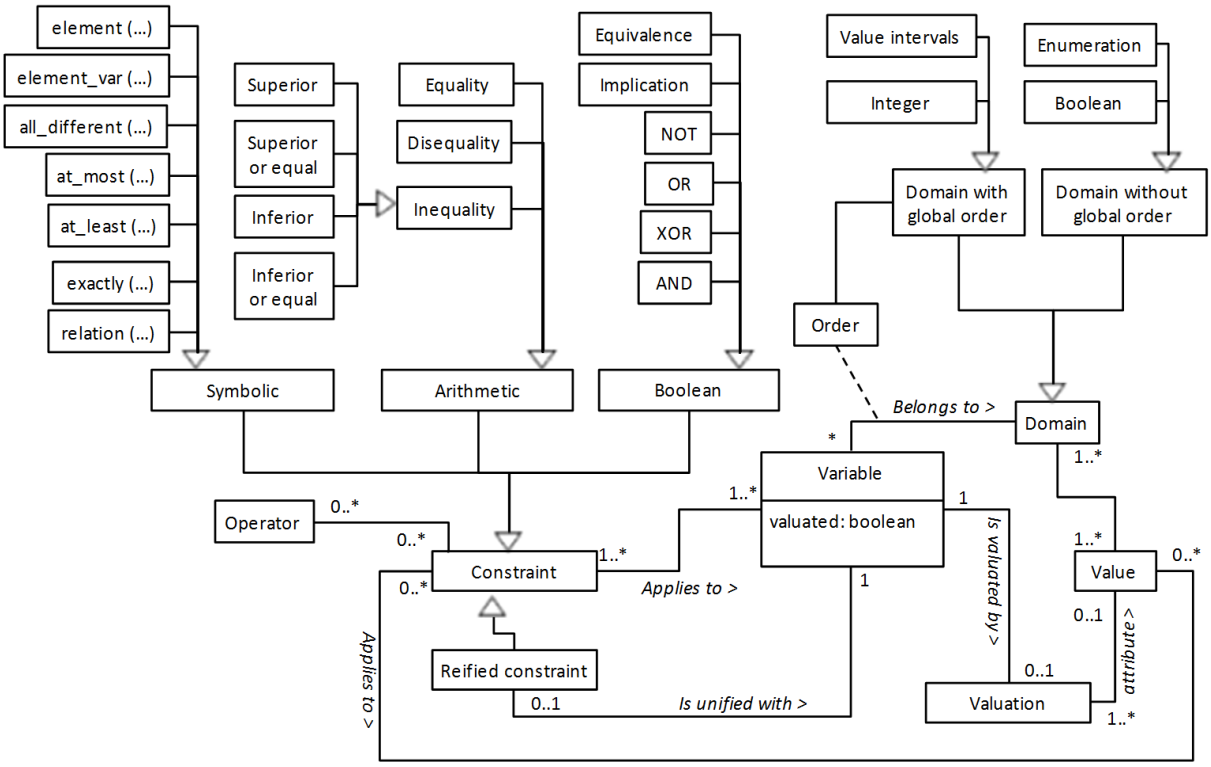

Figure 1. Meta-model of a constraint over finite domain language

Defining PL elements. Modelling PL using the CL consists in specifying constraints on PL elements (e.g., features, requirements, design fragments, 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 corresponding variable, where the 1 value means that the element is included in a configuration, and the 0 value means that it is not.

The statement `domain([`E_1`..`E_k`], 0, 1)` specifies that variables $E_1..E_k$ are Booleans. In the UNIX example, the graphical interface is specified with a boolean variable because it can be integrated or not in a UNIX operating system. This is specified by:

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

Also, it may be necessary to reason on the number of times a PL element can be repeated in a product, as suggested by (Czarnecki et al. 2005). In the UNIX example, a UNIX system can contain from zero to several thousands of processes. One may also deal with quantifiable elements such as performance, quantity or capacity.

This kind of constraints over variables that can appear several times in a configuration can be specified with a variable `E` with a finite domain `[m..n]`, `n` being for instance the maximum number of occurrences of `E`, or its maximum value or with an enumeration, as follows:

elements $E_1..E_k$ are integer elements: `domain([`E_1`..`E_k`], m, n)`

elements $E_1..E_k$ are enumeration elements: `domain([`E_1`..`E_k`],[`$value_1$`, ..., `$value_n$`])`

In the UNIX example, width resolution is represented by an enumerated variable:

```
domain([WidthResolution], [800, 1024, 1366])
```

Reasoning about Boolean variables. Basic and complex constraints can be specified over Boolean variables as follows.

Two elements E_1 and E_2 can only be either both present or both absent of a configuration: $E_2=E_1$

A configuration can contain an element E_2 only if it also contains E_1: $E_2 \leq E_1$

The elements E_1 and E_2 cannot be simultaneously included in the same configuration: $E_1+E_2 \leq 1$

A configuration can contain a number of at least Min (or at most Max) elements within a group of $E_1..E_k$ elements: $Min \leq \Sigma_{1..k} E_i$ and $\Sigma_{1..k} E_i \leq Max$

If E_3 is included in a configuration, then either E_1 or E_2 is included; otherwise all are excluded: $(E_3 \Rightarrow E_1 + E_2 = 1) \wedge (\neg E_3 \Rightarrow E_1 + E_2 = 0)$, or more concisely $E1 + E2 = E3$. For instance in the UNIX example (Constraint 11), Editing implies the inclusion of UserInterface. This can be specified by the constraint:
$Editing \Rightarrow UserInterface$

Given two sets of elements $S_1 = \{E_1, E_2\}$ and $S_2 = \{E_3, E_4\}$, a configuration should contain more elements from the set S_1 than from the set S_2: $E_1 + E_2 > E_3 + E_4$. This constraint can, of course, be extended to larger sets.

Either E_1 is included in a configuration, or both E_2 and E_3: $2 * E_1 + E_2 + E_3 = 2$

Multiple requires: if the boolean element $E_z \notin \{E_1..E_k\}$ belongs to a configuration, then the elements $\{E_1..E_k\}$ should be there too. The corresponding constraint is: $E_z \Rightarrow (E_1 \wedge .. \wedge E_k)$

Multiple exclusion: if the boolean elements $\{E_1..E_k\}$ belong to a configuration, then $E_z \notin \{E_1..E_k\}$ should be excluded. The corresponding constraint is: $(E_1 \wedge .. \wedge E_k) \Leftrightarrow \neg E_z$

Reasoning about integer variables. Basic and complex constraints can be specified over integer variables as follows.

$E_1 > a$: to indicate that element E_1 shall be included at least a times (i.e., it has at least a occurrences in a given configuration); if the variable represents an attribute, then the constraint means that its value shall be superior to a.

$E_1 = a$: to specify that the number of times element E1 can be included in a configuration is a fixed constant. For instance, if the variable Graphical = 0, then WidthResolution = 0 and vice versa.

$E_1 \neq a$: to indicate that element E_1 shall not be included a times in a configuration.

Multiple requires: if the integer element $E_z \notin \{E_1..E_k\}$ belongs to a configuration, then the integer elements $\{E_1..E_k\}$ should be there too. The corresponding constraint is: $(E_z > 0) \Rightarrow ((E_1 > 0) \wedge .. \wedge (E_k > 0))$

Multiple exclusion: if the integer elements $\{E_1..E_k\}$ belong to a configuration, then $E_z \notin \{E_1..E_k\}$ should be excluded. The corresponding constraint is: $((E_1 > 0) \wedge .. \wedge (E_k > 0)) \Leftrightarrow (E_z = 0)$

Mutual exclusion: elements E_1 and E_2 are mutually exclusive, that is, both are excluded or if one is included in a configuration, then the other should be excluded from that configuration: $E_1 * E_2 = 0$

A configuration should include more occurrences of an element than of another: $E_1 > E_2$

A configuration should include as many occurrences of an element E_1 as of two other elements (E_2 and E_3) together: $E_1 = E_2 + E_3$; this is for example useful to specify that a UNIX system may be installed by one of three methods presented in Constraint 8: $UNIX = Cdrom + Usb + Net$.

Numeric dependency: in the example, n additional kernels are needed for other purposes when UNIX is configured. This is specified by: $UNIX > 0 \wedge Kernel = n$.

A configuration should include more occurrences of a pair of elements (E_3, E_4) than of another pair of elements (E_1, E_2) together: $E_1 + E_2 < E_3 + E_4$.

The number of occurrences of E_1 should be the half of the number of occurrences of E_2: $2 * E_1 = E_2$.

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([`E_1`, .., `E_k`])`: specifies that in any configuration the value of each of the $E_1 \ldots E_k$ elements should be different pair wise.

`atmost(n, [`$E_1..E_k$`], a)`: specifies that at most `n` of the $E_1 \ldots E_k$ elements are equal to `a`.

`atleast(n, [`$E_1..E_k$`], a)`: specifies that at least `n` of the $E_1 \ldots E_k$ elements are equal to `a`.

`exactly(n, [`$E_1..E_k$`], a)`: specifies that exactly `n` of the $E_1 \ldots E_k$ elements are equal to `a`.

`relation([`$E_1..E_k$`],{[`$a_1..a_k$`]})`: constraints the tuple of elements `[`$E_1..E_k$`]` to be equal to at least one tuple in the collection of tuples `[`$a_1..a_k$`]`. This allows to specify extensively a predetermined collection of compatible values for a series of elements.

In the UNIX example (Constraint 8), symbolic constraints can be used to specify predefined combinations of the values that `WidthResolution` and `HeightResolution` can take in a particular configuration :

```
relation ([WidthResolution,HeightResolution], [[800, 600], [1024, 768], [1366, 768]])
```

Constraint Reification. In CP, the reification of a `constraintC` into a variable `C` of the `[0..1]` domain is achieved by a constraint:

$C \Leftrightarrow$ `contraintC`

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:

$E_1 = 1 \Rightarrow C$: whenever E_1 is included, the constraint `constraintC` reified with the `C` variable should be satisfied. Conversely, as soon as $\neg C$ is detected, E_1 is set to `0`.

$E_1 = 0 \Rightarrow C$: whenever E_1 is excluded, the constraint `constraintC` reified with `C` should be satisfied.

In the UNIX example, if the shell feature is selected in a particular configuration, then, several kinds of shells can be selected for each user, as presented in rule 10 of the user view:

$$Shell \Rightarrow ((1 * ROOT\_USER \leq ROOT\_USER_{SH} + ROOT\_USER_{TCSH} + ROOT\_USER_{BASH} \leq 3 * ROOT\_USER) \wedge (1 * GUEST\_USER \leq GUEST\_USER_{SH} + GUEST\_USER_{TCSH} + GUEST\_USER_{BASH} \leq 3 * GUEST\_USER))$$

Application on the example

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

Technical view:

$[UNIX, Kernel, Scheduling, ExecutingInstructions, InterpretingInstructions, AccomplishingTheTransferOfData, AllocatingTheMachine'sMemory, Shell, FileSystem, UserInterface, Graphical, Process_1, ..., Process_k] \in \{0,1\} \wedge$
$WidthResolution \in \{800, 1024, 1366\} \wedge$
$HeightResolution \in \{600, 768\} \wedge$
$[Support\_usb, Cdrom\_atech, Pcmcia\_support] \in \{0,1,2\} \wedge$
$UNIX = Kernel \wedge$
$(Kernel = AllocatingTheMachine'sMemory) \Rightarrow Process \wedge$
$(Kernel = Scheduling) \Rightarrow Process \wedge$
$(Kernel = AccomplishingTheTransferOfData) \Rightarrow Process \wedge$
$Shell \Rightarrow (Kernel = InterpretingInstructions) \wedge$
$Shell \Rightarrow (Kernel = ExecutingInstructions) \wedge$
$(UNIX = Process_1 \vee ... \vee UNIX = Process_k) \wedge$
$UNIX = FileSystem \wedge$
$(Support\_Usb > 0) \Rightarrow A \wedge$
$(Cdrom\_Atech > 0) \Rightarrow B \wedge$
$(Pcmcia\_Support > 0) \Rightarrow C \wedge$
$Kernel > 0 \Leftrightarrow (0 \leq A + B + C) \wedge$
$Kernel > 0 \Leftrightarrow (A + B + C \leq 3) \wedge$
$Shell \Rightarrow Kernel \wedge$
$Graphical = 1 \Leftrightarrow (WidthResolution = W1 \wedge HeightResolution = H1) \wedge$
$Graphical = 0 \Leftrightarrow (WidthResolution = 0 \wedge HeightResolution = 0) \wedge$
$relation([W1, H1], [[800, 600], [1024, 768], [1366, 768]])$

User view:

$[UNIX, UserInterface, Textual, Graphical, Cdrom, Usb, Net, UtilityProgram, Editing, FileMaintenance, ProgrammingSupport, OnlineInfo, Shell, ROOT\_USER_{SH}, ROOT\_USER_{TCSH}, ROOT\_USER_{BASH}, GUEST\_USER_{SH}, GUEST\_USER_{TCSH}, GUEST\_USER_{BASH}] \in \{0,1\} \wedge$
$UNIX \leq Cdrom + Usb + Net \leq UNIX \wedge$
$UtilityProgram \leq UNIX \wedge$
$Shell \Rightarrow UtilityProgram \wedge$
$Shell \Rightarrow ((1 * ROOT\_USER \leq ROOT\_USER_{SH} + ROOT\_USER_{TCSH} + ROOT\_USER_{BASH} \leq 3 * ROOT\_USER) \wedge (1 * GUEST\_USER \leq GUEST\_USER_{SH} + GUEST\_USER_{TCSH} + GUEST\_USER_{BASH} \leq 3 * GUEST\_USER)) \wedge$
$Editing = UtilityProgram \wedge$
$FileMaintenance = UtilityProgram \wedge$
$Printing \leq UtilityProgram \wedge$
$UserInterface \leq UtilityProgram \wedge$
$ProgrammingSupport \leq UtilityProgram \wedge$
$OnlineInfo \leq UtilityProgram \wedge$
$1 * UserInterface \leq Graphical + Textual \leq 2 * UserInterface$

Any constraint solver can be then used to solve this constraint program. We used GNU Prolog (Diaz, Codognet 2001) to analyze the UNIX system. With the technical view, we obtained a list of 100440 products that were generated (in 515 milliseconds CPU time) and with the user view, we obtained a list of 408 products (in 15 ms CPU time).

If the product line is already specified with a traditional formalism, then, it is possible to transform the models into CPs. For instance, Mazo et al. show how to transform Dopler and feature models, respectively, into CPs to verify and analyze them (Mazo et al. 2011a, Mazo et al. 2011d).

Mapping Between Constraint Programming Product Line Specification and Other Formalisms

CP is a paradigm that maps with numerous variability notations. Tables 2 to 6 in the appendix present the constrain representation of some common constructs different PL modelling formalisms.

Table 2 deals with FODA-like models (Kang et al. 2002), and Feature models with cardinalities (Riebisch et al. 2002, Czarnecki et al. 2005) and attributes (Benavides et al. 2005c, Streitferdt et al. 2006, White et al. 2009). While the constructs of the former FODA dialect can be mapped into Boolean variables and Boolean constrains, the constructs of the second one map into Boolean and Integer variables, and Boolean, Arithmetic, Symbolic and Reified constraints.

Table 3 deals with Orthogonal Variability Models (OVMs) (Pohl et al. 2005) and the Textual Variability Language (TVL) (Boucher et al. 2010). The constraints that map the constructs for OVM are boolean, arithmetic, symbolic and reified. All these constraints and several symbolic constraints like *sum, mul, min, max, avg* and *count* map with the TVL. The variables that map with elements and attributes from these two formalisms have Boolean, Integer and Real domains.

Table 4 presents the mapping between CP and the constructs of the Class-based (Ziadi 2004), (Korherr & List 2007) and the Use case-based (Van der Maßen & Lichter 2002) formalisms. The variables that map with elements of the Class-based language are boolean, integer and real. Those that map with elements of the Use case-based language are boolean. The constructs of both modelling languages are represented in CP as Boolean and Arithmetic constraints.

Table 5 presents the constructs of the Dopler variability language (Dhungana et al. 2010) and the variability language defined at the French *Commissariat à l'Energie Atomique* (CEA). The Dopler language is already semantically rich as it uses the Java constraints and domains. We therefore suggest to use the same kind of constraints and variables to represent the constructs of Dopler models with CP. The constructs of the CEA language map with Arithmetic and Boolean constrains over boolean variables.

Finally, Table 6 presents the mapping between CP and constructs of the Renault Documentary Language (RDL) and the Latice language proposed by (Mannion 2002). In both cases, all the constructs map with Boolean constraints over Boolean variables.

The lessons learned from these mapping rules can be summarized as follows:

A. All the classic or industry-specific product line modelling formalisms that we considered in this study could be represented as constraint programs.
B. The same constraints have different representations in the different formalism, and even different representations in the same formalism. For instance in feature-based languages, an exclusion constraint can be represented by means of the exclusion relationship or by an XOR construct.
C. There are constraints that do not map with any construct in any of the product line modelling formalism considered in the study. For instance, "at least n of the $E_1 \ldots E_k$ elements are equal to a" can easily be specified with CP while it can neither with FODA dialect nor with OVM, TVL, UML-based, or DOPLER.

An interesting observation is that being notation-independent, the constraints presented in the tables shown in the appendix can be compiled in any of the languages of the off-the-shelf solvers in which they shall be executed. This way, one can exploit the best characteristics (performance, functions) of each off-the-shelf solver to execute each analysis or configuration operation. We propose, on the one hand, to use Constraint Logic Programming (CLP) solvers to deal with PLMs

containing boolean and non-boolean variables, which is very common, for instance, in feature models with attributes (Streitferdt et al. 2003), (White et al. 2009). On the other hand, when the model has only Boolean variables, if the goal is to compute the number of configurations, we suggest to use SAT-based model counters (or possibly Linear and Binary Decision Diagram solvers even if they have well known limitations when variables are entered in a wrong order; cf. Mendonça et al. 2009b). Indeed, these kinds of solvers are designed to efficiently calculate the number of solutions of constraints programs (Mendonça et al. 2009b). In addition we propose to use CLP solvers for models with non-boolean variables (integer, real, etc.) and SAT or Satisfiability Modulo Theories (SMT) solvers for models with only Boolean variables. Indeed, SMT solvers show good performances when executing satisfiability operations on arithmetic constrains over Boolean variables.

Representing PLMs as constrain programs over different domains allows taking advantage of the best characteristics of the different existing solvers. This idea is summarized in the framework presented in Figure 2. In the figure, not only PLMs but any kind of variability models are represented as constraints with a unique CP notation that encompass other constraint languages (e.g., over Booleans, Integers, Reals, trees, lists, etc.). Therefore, the CP language acts as an interoperability notation as (a) it is able to deal with different meta-models, (b) it helps deal with several models at the same time, as we show it in the following sections, and (c) it can be executed with different solvers that can be chosen depending on the context.

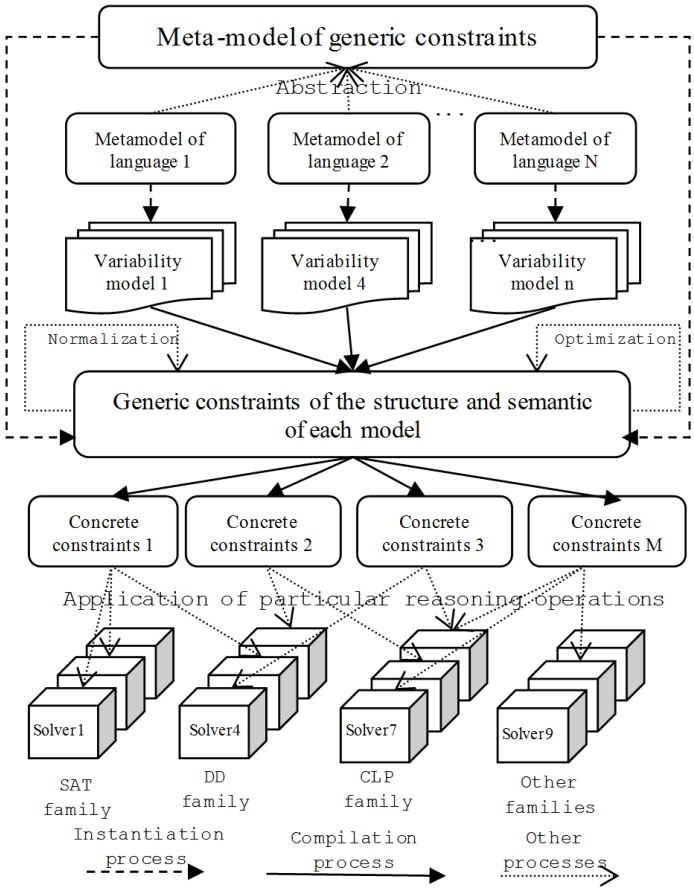

Figure 2. Generic constraints to represent variability models.

As Figure 2 shows it, once variability models are specified as generic constraints, they can be transformed into concrete constraints. By “generic” we mean platform independent (Saraswat 1992). This means that wherever they come from, constraints can be compiled with the platform into any constraint language. The language and associated solver are chosen depending on the analysis to achieve and expected performance.

INTEGRATION OF CONSTRAINT-BASED PRODUCT LINE MODELS

An important challenge in PL domain engineering and application engineering is that product lines are often, in practice, specified using several models at the same time (Djebbi et al. 2007, Segura et al. 2008, Rosenmüller et al. 2011). As when describing the architecture of any kind of system, this allows dealing with various facets of the PL and products, and representing the viewpoints of various stakeholders such as executives, developers, distributors, marketing, architects, testers, etc. (Nuseibeh et al. 1994). For example, analysts may deliver a requirements model that specifies user-oriented system functionality, while architects may deliver a feature-based model focusing on the system structure from a more technical design-oriented point of view. In the absence of a global model, and given the number models in which the PL can be specified, requirements can get missed or misunderstood (Finkelstein et al. 1992) both during domain and application engineering activities. There are 2 other problems related with multi-model PL specifications: inadequate support for multi model specification, and weak support for the maintenance of the global PL specification.

The size and complexity of industrial product lines motivates the specification of PL models by heterogeneous teams (Dhungana et al. 2006), (Segura et al. 2008). However, existing tools provide only little support to integrate multiple models and to perform the analysis and configuration activities on the global level. To our knowledge, there is no proposal so far to integrate PL models specified with different formalisms.

Besides, it is a fact of industrial life that product line models evolve over time, for instance to reflect new marketing requirements, product level innovations that should be capitalized at the PL level, or new design decisions about the PL architecture. The problem is that any change in a model can impact other models too. For example, changes in the architecture can make the corresponding model inconsistent with the technical solution models, or with the PL models that represent the sales and marketing viewpoints. To the best of our knowledge, there is no tool that provides automated mechanisms for analysing the impact of changes of a PL model onto another one, or for ensuring the global consistency of changes achieved on multi-model PL specifications.

CP can be exploited in the context of multi-model PL engineering to capture in a unified way the various models, and to arrange them into a unique specification. As a result, domain and application engineering activities such as PL analysis or product configuration are facilitated. Indeed, the unique representation facilitates the propagation of constraints between variables that belong to the different models. When configuration entails a variable in a model, it entails the variable in all the other models to which the variable belongs.

Another considerable advantage is that having all the models of the PL integrated in a single CP allows specifying constraints between different variables that belong to different models. Our literature survey did not reveal any interoperability meta-model that would have allowed relating several PL models as proposed here.

Motivated by the pertinence of the subject and the requirements of our industrial partners, we developed a constraint-based integration process for product line models. In our process, integrating two PLMs consists in (i) integrating the variables that correspond to reusable elements; (ii) integrating attributes and their domains and; (iii) integrating the relationships among

reusable elements. Integrating two models can be done in two steps: matching and merging (Finkelstein et al. 1992), (Fleurey et al. 2007). The matching step specifies which element in the language can match and how they can match. The merge step defines, how two model elements that match are merged, as well as a mechanism to handle the non-matching elements of the input models. For example, if two feature models (Kang et al. 1990) that specify a single PL own the same feature A, which is being required by another feature in the first model, and which is excluded by another feature in the second model, then the situation match because of the feature A. However, the decision to include or not feature A in the resulting model depends of the merging rules and the integration strategy. In particular, one has to reason on the dependencies between feature A and the other features in the two models.

Integration strategies are about the ways in which models are merged. Indeed, different merging rules exist and may be used in given matching situations. One scenario can be, for instance, when a company decides to lengthen the production spectrum of the PL, and therefore integrates the PLMs of two headquarters and keeps in the resulting PLM the reusable elements and the production capacity of both headquarters. We identified five different strategies that may be used to integrate PLMs: two restrictive strategies, two conservative strategies and one disjunctive strategy.

Strategy N° 1 is restrictive in the sense that it allows representing in the resulting PLM the common products represented in both input models that can be configured with the common reusable elements and attributes.

Strategy N° 2 is also restrictive, but differently from the first one: the products can be configured with all reusable elements and attributes available on both input models (Hacher et al. 2010).

Strategy N° 3 is conservative in the sense that it allows configuring the products represented in both input models by using only the common reusable elements and attributes.

Strategy N° 4 is also conservative but this time allows configure products with all reusable elements and attributes available in both input models (Segura et al. 2008), (Hacher et al. 2010).

Strategy N° 5 is disjunctive in the sense that the resulting model allows configuring the products presented on one of the input models by using the reusable elements and attributes of one of the particular models but not these of the other one.

We propose 89 constraint-based integration rules for product line models (Mazo 2011); these rules are not presented here because it is out of scope of this paper. However, one example of each integration strategy is presented in Table 1. The example used to illustrate the five integration strategies consists in a mandatory relationship between variables `A` and `B` in the first model and an optional relationship between variables `A` and `B` followed by an implication to variable `C`. As is presented in Table 1, for the strategy 1 we only keep the variables present in both models and we related them with the most restrictive constraint; that is, we use the equality constraint instead of the superior or equal inequality due to the fact that the first one is more restrictive than the second one.

Table 1. Example of application of the 5 PLM integration strategies

Strategy	Base model 1	Base model 2	Resulting model
1	$A = B$	$(A \geq B) \Rightarrow C$	$A = B$
2	$A = B$	$(A \geq B) \Rightarrow C$	$(A = B) \Rightarrow C$
3	$A = B$	$(A \geq B) \Rightarrow C$	$A \geq B$
4	$A = B$	$(A \geq B) \Rightarrow C$	$(A \geq B) \Rightarrow C$
5	$A = B$	$(A \geq B) \Rightarrow C$	$(A = B) \oplus ((A \geq B) \Rightarrow C)$

To illustrate the fact that the equality constraint is more restrictive that the superior or equal inequality, please consider the two simple cases `A = B` and `A ≥ B`. Assuming that `A` and `B` are boolean variables; in the first case we can generate two configurations: `Conf1= {A=0, B=0}, Conf2= {A=1, B=1}` and in the second case we can generate three configurations: `Conf1= {A=0, B=0}, Conf2= {A=1, B=0}, Conf3= { A=1, B=1}`. The other two examples shown in Table 1 apply the same reasoning based on the aforementioned integration strategies.

VERIFICATION OF CONSTRAINT-BASED PRODUCT LINE MODELS

The use of constraint programming for software verification is not new (Collavizza & Rueher 2006), nor is the verification of product line models (Benavides et al. 2005, Trinidad et al. 2008, Salinesi et al. 2010, Mazo et al. 2011a). Verification of product line models consists in finding errors on these models. As in the case of analysis of product line models, these models must often be specified with a formalism that allows automatic verification (Batory et al. 2001, Benavides et al. 2005, Karataş et al. 2010, Mazo et al. 2011d). Automatic verification of product line models is highly needed: indeed their manual verification is an error–prone, tedious and sometimes infeasible task due to the complexity of these models (Benavides et al. 2005), (Trinidad et al. 2008), (Salinesi et al. 2010).

Verifying PLMs entails several aspects. On the one hand, a product line model, independently of the formalism used to specify it, must respect certain properties associated with the domain of product lines. On the other hand, certain properties are associated with the fact that each PLM respects the syntactic rules of the formalism in which it is specified. Therefore, some properties of PLMs are independent of the formalism while other ones are particular to each formalism.

In light of this observation, we propose a typology of verification criteria (Salinesi et al. 2010a) that is summarized in Figure 3. This typology shows that not all criteria are equivalent. Some result of the formalization of the PL with a model (conformance checking; c.f., Mazo et al. 2011c), whereas others can be used to verify PLMs independent of their metamodel (domain-specific verification). Our experience with both kinds of verification shows that constraint logic programming (in this framework, constraints are embedded in the logic programming paradigm (Apt & Wallace 2006)) can be used to verify both kinds of verifications. In the context of domain-specific verification, there is a collection of verification criteria that every PLM must respect, independently of the formalism in which the model is specified. For instance, every PLM must allow configuring several products; that is to say the PLM must not be a void model. Our research have also demonstrated that in the context of conformance checking the criteria are not generic; on the contrary, they depend on the metamodel of the language in which each PLM is represented (Mazo et al. 2011b). For instance, a feature model must not have more than one root feature or must not have two features with the same name.

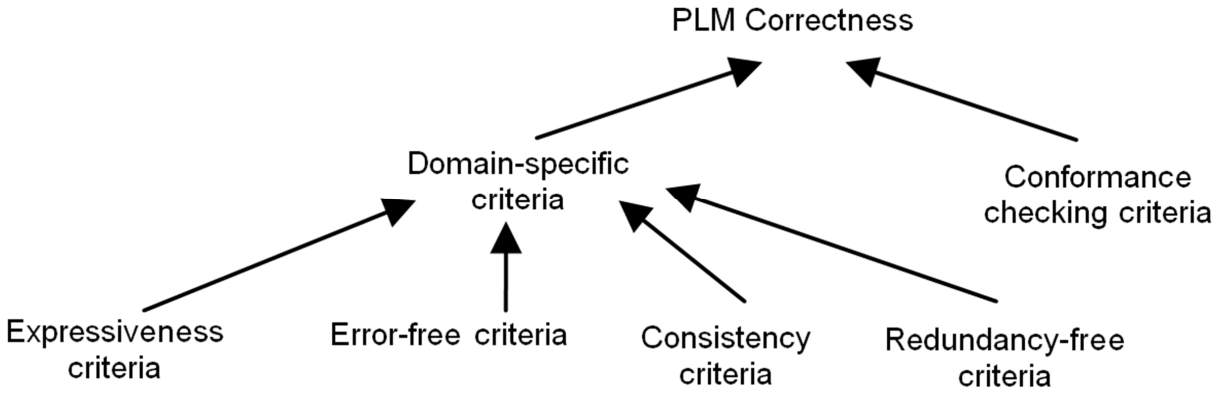

Figure 3. Typology of verification criteria on PLMs

The outcomes of the typology are multiple:

A. each defect can be searched for using a given criterion;
B. the typology facilitates the identification of defects for which no verification criterion is available elsewhere in the literature (Mazo et al. 2011a);
C. the classification behind the typology makes it easier the proposition of a standard and reusable approach to verify the domain-specific criteria of PLMs; and
D. the typology can be used to select the criteria that one wants to use to verify a PLM according to the impact that these criteria have or the expected level of quality of a particular PLM.

An example of consistency criteria is the absence of *false optional* variables on PLMs. In our running example, `UserInterface` is a false optional variable. Indeed, the variable is specified as optional but it is in fact present in all products of the PLM.
Another example of criteria is the following one: a PLM is satisfiable or *not void* if at least one product can be configured from the PLM. A void PLM is a model that does not allow configuring products; it is thus a useless PL model.

ANALYSIS OF CONSTRAINT-BASED PRODUCT LINE MODELS

Analysing product line models consists in the extraction of information from these models. As shown in Figure 4, specifying PL models with CP allows automatic analysis at both the domain and the application levels. When analysis is done at the domain level it helps reasoning on the PL itself. When achieved at the application level, it helps reasoning about products, mainly by generating either partial or full configuration (which corresponds to a valuation of some/all the variables). To our knowledge, the most extensive collection of analysis operations on feature models is the one presented in (Benavides et al. 2010). Most of these operations are discussed in this section.

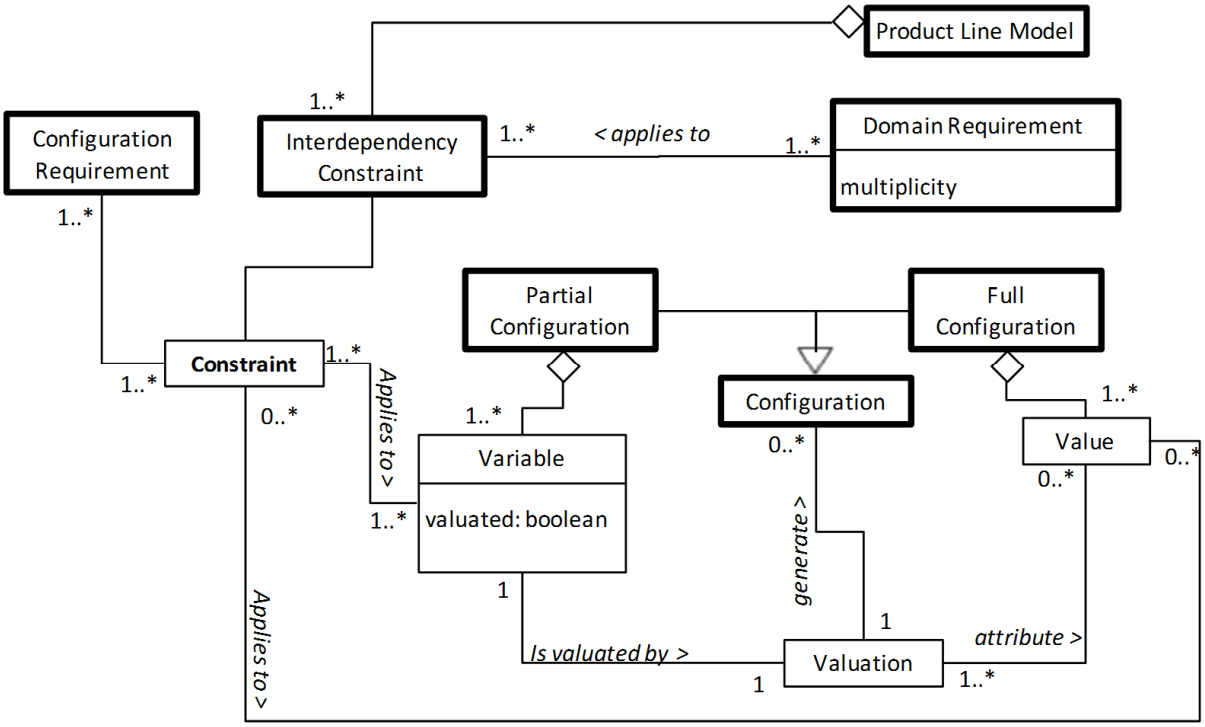

Figure 4. Product line analysis using the Constraint Language (Salinesi et al. 2011).

Domain Level Analysis.

Domain level analysis is performed on the PL itself, and not on the configured products. Some common operations on the domain level are:

1. Calculating the *number of valid products* represented by the PLM. This operation may be useful to determine the richness of a PLM. For instance, in our UNIX example, we obtain a list of 100440 products in the technical view and 408 products in the user view.
2. Calculating *commonality* of a collection of variables. This is the ratio between the number of products in which a collection of variables (e.g., a configuration) is present and the number of products represented in the PLM.
3. Calculating *homogeneity*: this indicates to which degree the elements appear in various products. A more homogeneous PLM would be one with few unique variables (a unique variable equals 1 only in one product) while a less homogeneous PLM would be one with a lot of unique variables. By definition `Homogeneity = 1 - (#unicVar / #products)` where `#unicVar` is the number of unique variables in one product and `#products` denotes the total number of products represented by the PLM. In our running example *Homogeneity* is equal to 0,99998.
4. Calculating *variability factor*: this operation returns the ratio between the number of products and 2^n where n is the number of variables considered. In particular, 2n is the potential number of products represented by a PLM, assuming that there are not transverse dependencies (in the sense of FODA) in the model and that all PLM's variables are boolean. `Variability factor = NProd / 2`^{n}. This function is not applicable to our UNIX's technical and user views because these models have integer variables and a lot of cross-tree constraints.

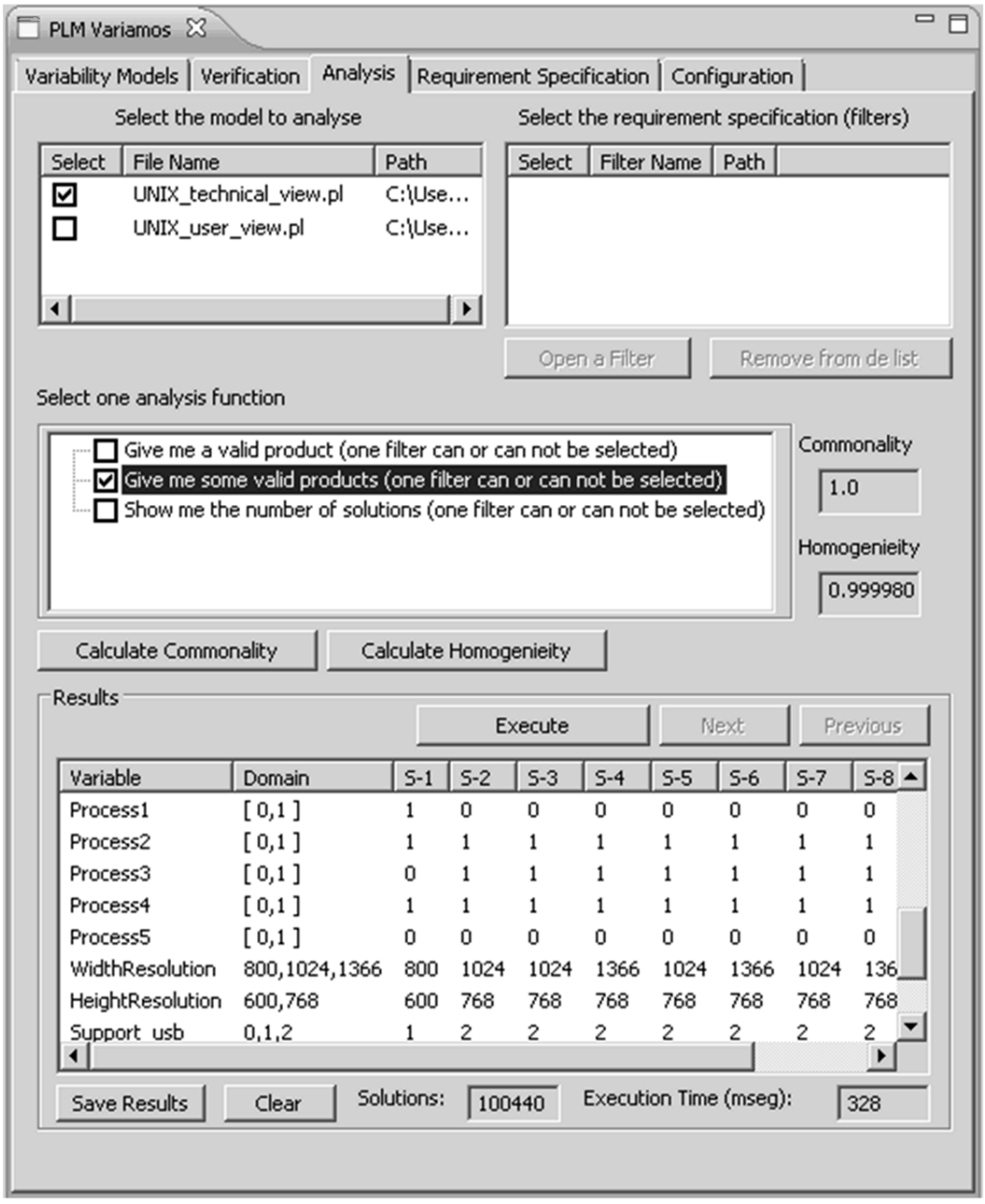

Figure 5. Some analysis functions over our UNIX's technical view, using our tool VariaMos

Application Level Analysis

The analysis operations at this level are the following ones:

5. *Finding a valid product*, if any. A valid product is a configuration that respects all the constraints of the PLM. For instance, finding a valid product configured with the UNIX technical view model is (as shown in Figure 5):

```
P1 = {UNIX=1, Kernel=1, Scheduling=0, ExecutingInstructions=1,
InterpretingInstructions=1, AccomplishingTheTransferOfData=0,
```

`AllocatingTheMachinesMemory=1, Shell=1, FileSystem=1, UserInterface=1, Graphical=1, Process1=1, Process2=1, Process3=0, Process4=0, Process5=1, WidthResolution=1024, HeightResolution=768, Support_usb=0, Cdrom_atech=2, Pcmcia_support=2}`.

It is worth noting that for boolean variables like `Graphical` the value `1` means that the corresponding element is present in the product, for `WidthResolution` the value `1024` represents the number of pixels corresponding the to the width resolution of the configured product, the value `2` of variables `Cdrom_atech` and `Pcmcia_support` means that these modules are static in the product, and the value `0` affected to `Support_usb` means that this module is not selected in the product.

6. Obtaining the list of *all valid products* represented by the PLM, if any exist. This operation may be useful to compare two product line models. For the sake of space, the comprehensive list cannot be presented in this paper, but as the screenshot shows it in Figure 5, tool support provides users with the possibility to navigate in the list of products using the *Next* and *Previous* buttons.

CONFIGURATION OF CONSTRAINT-BASED PRODUCT LINE MODELS

Product configuration is hard because of the quantity of product line elements, of their diversity and of the complex interdependencies between them. This problem becomes more complex when the product line is represented by multiple views, as in the case of our running example. Furthermore, customers typically have requirements that cannot be fulfilled by the product line. Also, they are lost with the number of choices and they find it difficult to find a product that belongs to the PL because they do not take configuration constraints into account when they specify their requirements. Therefore, automated mechanisms to propagate configuration decisions and guide the user in the configuration process are highly needed. Our experience with configuration of product line models (Djebbi et al. 2008), (Djebbi & Salinesi 2008) shows that constraints play a prominent role in the configuration process: in fact, stakeholders have a much richer power of expression when they use constraints to specify their configuration requirements. Our position is that configuration requirements should be considered as a first class concept. We have discovered that various kinds of configuration requirements can be specified using CP. Based on our observations we distinguish between three categories of configuration requirements: simple full closure, optimization, and partial closure.

Simple full closure: these simple full closure configuration requirements consist in allocating a given value to a variable. This kind of requirement is the most common one, as traditional PL specification formalisms are usually used in configurations to indicate whether or not a configuration should include a feature, a variant, etc. In CP terms, this kind of requirement is simply specified with a constraint `V = val`, where `val` is one of the possible values in the domain of `V`.

Optimization requirements consist in indicating that the configuration should be optimal in terms of one of the variables that define it (if the variable can be maximized of minimized). Examples of optimization requirements are maximization of revenue, of performance, or minimization of cost, delivery time, response time, etc. Optimization requirements can be combined, but this raises a hard problem as it may be difficult (i.e. NP complete from a calculability point of view) to satisfy all the requirements at the same time. Current researches on CP try to solve these issues by using smart strategies. Two approaches can be used at the PL levels. On the one hand, priorities can be used to specify which optimization requirement should be satisfied first. The other approach consists in proceeding in an incremental way by lowering the level of expectations for some of the optimization by specifying partial closure requirements.

Partial closure requirements are specified with constraints that reduce the possible list of values of one or several variables that specify the PL. For example a partial closure requirement can be specified to indicate close to optimal values for an attribute. Partial closure requirements can be complex in the sense that they may involve several variables at the same time. Of course, they can be simple too, as e.g. `WidthResolution ≥ 800` which involves one variable only. Examples of partial closure requirements are preferences (e.g., the user prefers `X` over `Y`), dependencies (e.g., if `X` is included in the configuration, then `Y` is not needed), and open choices (e.g., the configuration shall include at least 3 instances of `X`).

The following paragraphs present some configuration requirements patterns that are implemented in the VariaMos tool (Mazo et al. 2012). The following list shows some of the configuration requirements patterns and operations that can be performed with VariaMos to deal with configuration requirements.

1. *Optimal product*: `maximize(WidthResolution+ HeightResolution)` allows to find a solution such that the objective function `WidthResolution+ HeightResolution` is maximized. Conversely, `minimize(WidthResolution+HeightResolution)` allows to find a product with the minimum resolution allowed by the product line model.
2. *Global optimization requirements*. Examples of global 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). Detection of "optimal" products is very important for decision makers as presented in (Djebbi & Salinesi, 2007, Salinesi et al., 2010b). Specifying them may require the specification of additional variables that did not exist in the PL specification.
3. *Preselected configuration*. Configurations may be partial or total. A valid partial configuration is a collection of variables that respect the constraints of the PLM but not necessarily representing a valid product (some variables of the PL are not valuated). A total configuration is a collection of variables values that respect the constraints of the PL specification, and where there is no variable that needs to be valued to specify a valid product. An operation that helps specifying partial pre-selected configurations may be useful to determine if there are not contradictions in a configuration requirement. In our running example, the product:

   ```
   P2 = UNIX=1, Kernel=_, Scheduling=_, ExecutingInstructions=_,
   InterpretingInstructions=_, AccomplishingTheTransferOfData=_,
   AllocatingTheMachinesMemory=_, Shell=_, FileSystem=_, UserInterface=_,
   Graphical=0, Process1=_, Process2=_, Process3=_, Process4=_, Process5=_,
   WidthResolution=1024, HeightResolution=768, Support_usb=_, Cdrom_atech=2,
   Pcmcia_support=_}
   ```

   Which is configured in our tool VariaMos as shown in Figure 6 (note that the symbol "_" assigned to a variable means that there is not a predefined value for the variable).

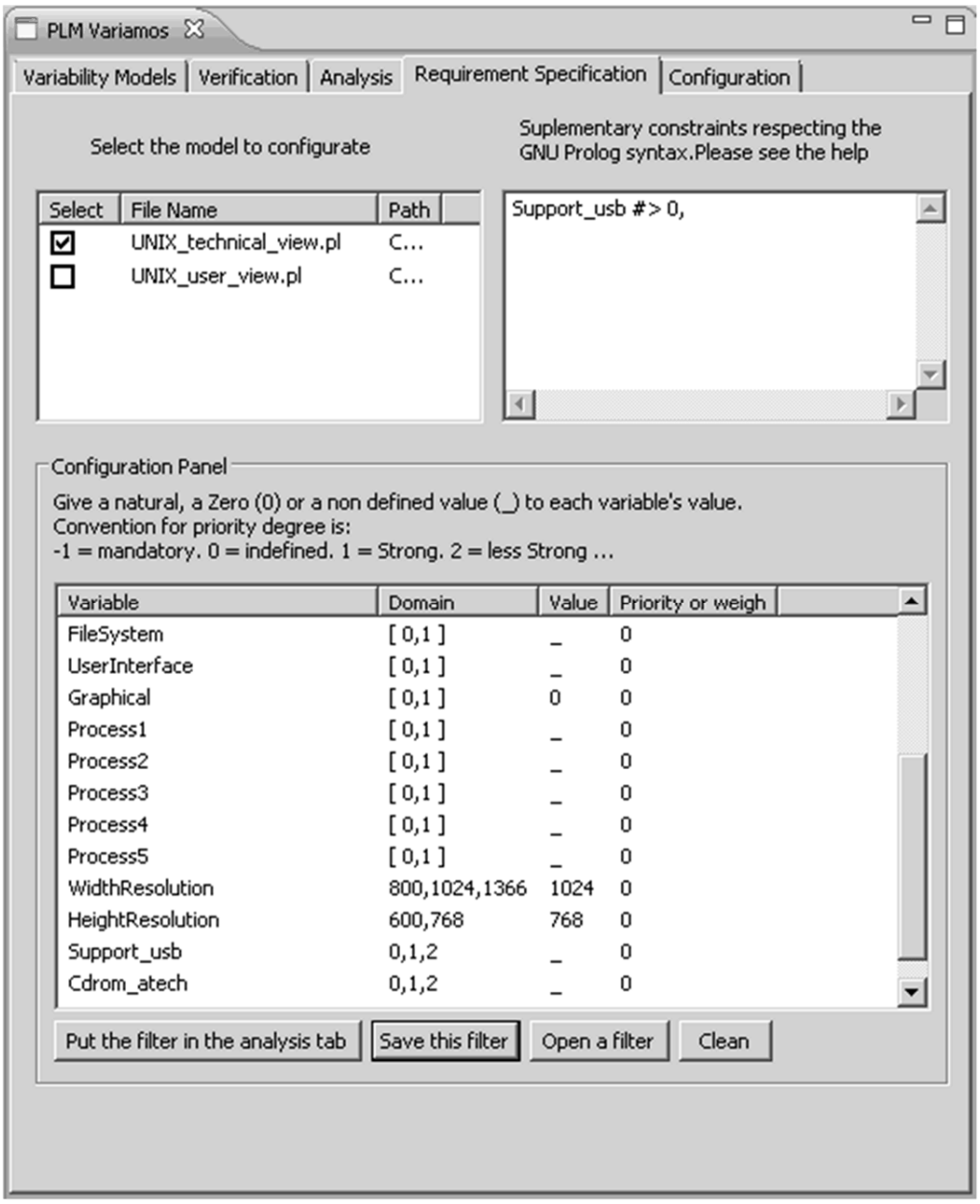

Figure 6. Requirement specification in order to create a filter with a certain configuration and supplementary constraints.

4. *Propagating dependency requirements*. The purpose is to look for all the possible solutions after assigning some fixed value to a collection of variables. In our running example, if one selects the `Support_usb` (that is, by assigning the value of 1 or 2, to the `Support_usb` variable), the variables `Kernel`, `UNIX` and `FileSystem` must be selected as well because of the constraints: $\texttt{UNIX} = \texttt{Kernel} \wedge \texttt{UNIX} = \texttt{FileSystem} \wedge (\texttt{Support\_usb} \leq \texttt{Kernel}) \Rightarrow \texttt{FileSystem}$. The number of products that satisfy this requirement is 66960.
5. *Filter*. This operation takes a partial configuration (i.e. a set of valued variables), the PL constraints specification, a collection of supplementary constraints that specify various kinds

of requirements. In return, it generates the collection of products that include the input partial configuration and respect both the constraints of the PLM and the configuration requirements.

6. *Calculating the number of products after applying a filter*. This is useful when too many products can be configured to navigate between. If we apply a filter constraining products with a resolution of `1204x768` and products where `Support_usb > 0`, there are 22320 correct configurations, which indicates to the stakeholder that further requirements are needed to come up with a manageable collection of configurations.

EVALUATION

We evaluated the effectiveness of our approach by testing its implementation, scalability and feasibility.

Tool implementation. We developed an interactive environment composed of two tools: VariaMos (Mazo et al. 2012) and GNU Prolog (Diaz & Codognet 2001). VariaMos is an Eclipse plug-in that allows managing constraint programs (e.g., creating, editing and saving them) to specify PL models. Besides, VariaMos offers rich collection of PL engineering functions such as verification (detect void models, false product line models, dead variables, false optional variables, not attainable domains and redundant constraints), analysis (supporting most of the aforementioned analysis functions) and configuration requirements specification (e.g., configure a product, define a filter or a partial configuration and specify extra constraints or particular requirements).

Computational Scalability. We assessed the scalability of our approach with 50 models, out of which 48 were taken from the SPLOT repository (Mendonca et al. 2009a). The other two models were developed during industry collaboration projects (Djebbi et al. 2007), (Lora-Michiels et al. 2010). The size of the models are distributed as follows: 30 models contained from 9 to 49 variables, 4 from 50 to 99, 4 from 100 to 999 and 9 from 1000 to 5000. The PL covered various domains such as insurance, entertainment, web applications, home automation, search engines, and databases. Note that the original SPLOT models only contained [*m..n*] cardinalities with *m* equal to 0 or 1, and they did not contain any attribute. Therefore, in order to increase the complexity, numerical attributes (such as `WidthResolution ∈ {800, 1024, 1366}`) were introduced in a random way, so as to have models with attributes associated with 30%, 60% or 100% of the other elements. Following the same logic, we changed 50% of the SPLOT cardinalities in order to have more general cases than the original ones. In order to do that, we created a simple tool[3] that translates models in the SPLOT format into constraints programs. This was achieved using the transformation patterns presented in (Salinesi et al. 2011), and by assigning artificial attributes and lower bound cardinalities in a random way as long as $m \geq n \geq 0$. The evaluation was performed in the following environment: desktop Intel Pentium 4 3.2 GHz PC with Windows Seven 32 bits, 4,00 GB RAM memory, GNU Prolog 1.3.0. The evaluation results are shown in Figure 7. The results are presented in a logarithmic scale for the sake of readability of the data distribution.

The experimental results presented in Figure 7 indicate that PLMs can be analyzed in an acceptable time. The best results from the point of view of scalability are obtained on the

[3] opt_semantic_parser_sxfm.jar available at: https://sites.google.com/site/raulmazo/

analysis operations. We were able to avoid computing all configurations to perform analysis functions by using a CLP solver to execute all the analysis operations discussed in the previous section. This kind of solver is not optimized to compute the number of solutions of a CP. However, this kind of solver (and SAT-based solvers too) is designed to be efficient on satisfiability operations like "check if the PLM is void or not". Owing to this, the worst computation times of VariaMos are 3,5 ms (c.f., Figure 7(a)) to test void models, 1.6 sec. (e.g., Figure 7(d)) to calculate the "variability factor", and 1,8 sec. (c.f., Figure 7(e)) to execute the "validate a configuration" operation.

Interestingly, Figure 7(g) shows an abnormal behaviour for a model of 89 variables. The time to calculate the whole collection of configurations from this model is very high: 15 min. The difficulty to analyze certain product line models (with a very high variability factor or with a very large number of cross-tree constrains) is treated in detail in (Mendonça et al. 2009). We have not yet found a systematic solution to overcome this issue.

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 applied our CP over Finite Domains (FD) approach to specify constraints on a family of blood analysis automatons (Djebbi & Salinesi 2007) in the context of a cooperation with the STAGO industry partner.

Using FD constraints allowed us to specify constraints to reason about cost and revenue of features of the STAGO instruments in the PL. In order to do this, we associated to each feature, two variables to specify costs and benefits respectively. 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 complex dependencies that could not be specified in FODA dialects. For instance "the optional function 'Agitate' shall be implemented if one of the tests `TCA`, `ATIII` or `PC` are not included in the configuration".

```
(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 could 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 ≥ LauchTest.TestCadence*
  LauchTest.TestDuration
  ```

- Some symbolic constraints such as `Atmost(1,[Agitate,Mix,Incubate],2])` which specifies that each activity in a methodology can be repeated at most twice, could have been specified with FODA, but this was so difficult, error prone and not flexible, that most of them were left apart.

- Other symbolic constraints could simply not be specified with FODA or its dialects. This is for example the case of the following symbolic constraints that was needed to specify possible combinations of value of the cadence, duration, and kind of determination for different kinds of test types:
  `Relation ([LauchTest.TestType, LauchTest.TestDuration, LauchTest.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 needed to specify reified constraints such as:
  `LaunchTest.TestType = TCA ⇔ C ∧ C ⇒ Chronometric=1 ∧ Chronometric.Speed = normal`
  which enforces the use chronometric measurement technique when TCA test is demanded, and specifies the required speed for this test. This constraint could not be specified using FODA dialects.

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)
```

The results obtained with the STAGO product line are encouraging and confirm that CP over FD is well suited to precisely model and efficiently configure PL.

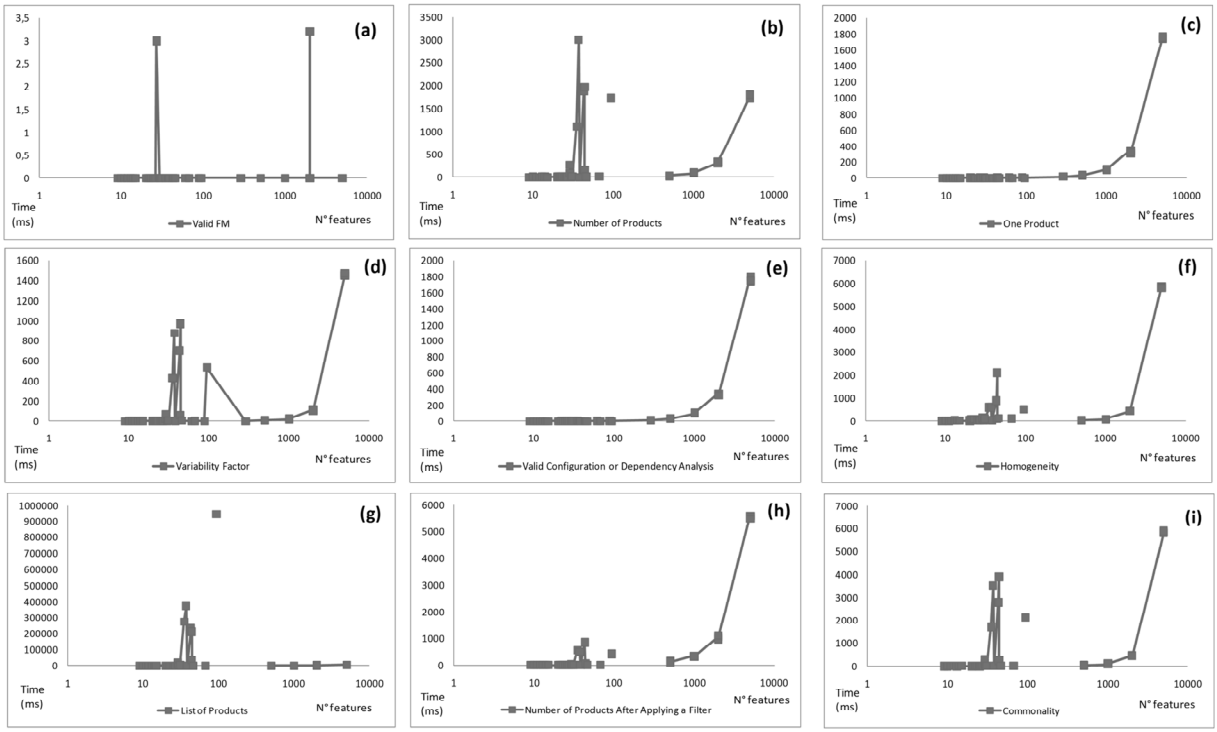

Figure 7. CPU time to execute our analysis operations over our 47 PLMs. Axe Y: time in milliseconds. Axe X Log 10 (Number of features)

Usability. This paper does not address how to best visualize our constraint-based approach neither the results obtained from the constraint-based product line models. Even if these issues are very important due to the fact that in some cases these models exceed hundreds of constraints and millions of products, much of this problem has to do with human-computer interaction. However, we propose a graphical representation of the constraint-based PLMs as a constraint network and a friendly user interface to present the results obtained from our analysis operations over the models. In the first case, the constraint network is a graph where each node corresponds to a constraint and each arch corresponds to the relationships among the variables of each constraint. For instance, in our running example, `UNIX = Kernel` is a constraint and therefore it is graphically represented as one node (`N1`) and `(Kernel = Scheduling)` $\Rightarrow$ `Process` is another constraint and therefore it is graphically represented as one node (`N2`). Due to the fact that `N1` and `N2` share one variable (`Kernel`), there is one arch between `N1` and `N2`. In the second case, we present the results step by step by means of matrix that contains collections of results (ten by ten) and the user can interact with the tool to configurated the number of solutions that he/she want to have in the screen.

Another important issue that is not addressed in this paper is the downstream economic benefits. For example, one could raise the question how does analysis operations really benefit software engineering at large? How much does it cost if such an analysis approach does not exists in a PL production environment? These complex issues have yet to be investigated.

RELATED WORKS AND DISCUSION

This paper is not the first one to explore the use of constraints programming in the context of PL. For instance, Mannion (2002) and Zhang et al. (2004) use propositional logic to represent PLMs, and Batory (2005) uses Conjunctive Normal Form (CNF) formulas to represent FMs and SAT solvers to analyze them. In these formulas, 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 (Benavides et al. 2005 and 200) 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. (2009) 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 (White et al. 2009) or successive partial configurations as in (Czarnecki et al. 2005).

More recent works (Karataş et al. 2010, Salinesi et al. 2010, Mazo et al. 2011d) show that specifying PL in CP rather than the original graphical language is not pertinent just because it facilitate the automatic reasoning on them but also because it increases the power of expression. Indeed, the drawback of graphical formalisms is that their power of expression is largely reduced because of the graphical notation. On the other hand CP has the advantages of the text based languages to represent and support reasoning on PLMs in an efficient way. The drawbacks of the graphical PL notations were well demonstrated by Heymans, Boucher and Classen who proposed a Text-based Variability Language (Boucher et al. 2010) and (Classen et al. 2011) to overcome

them. Some of these drawbacks are: (i) to create a large PLM with a graphical syntax “is a burden and cannot be mastered without dedicated tool support”, (ii) given the tree and two dimensional structure of most of the PL notations “there will inevitably be large physical distances between features, which makes it hard to navigate, search or interpret the model”, and (iii) “most notations do not have graphical means to represent constructs like attributes and constraints which are essential for industrial FMs”. On the other hand, the advantages of non-graphical languages are: (i) the expressiveness to represent variability and commonality constructs that are forbidden in some graphical notations due to the limited syntax of the these languages; and (ii) the possibility to reason directly on the model (with, in some cases, a compilation instead of transformation to a low-level language). This latest property avoids problems related to loss of information and misinterpretation when PLMs are transformed from their original formalisms to an executable language. In addition, our experience has shown that both, structure and semantic of product line models can be represented by means of logic and constraint constructs.

Besides, the aforementioned approaches consider only single monolithic feature models. As shown Tables 2 to 6 in the appendix, our approach is able to deal with several models including when they are specified using different formalisms. Furthermore, our approach explores more FD Constraint Programming capabilities that have not been exploited so far. For instance it provides numerous types of constraints (e.g. symbolic and reified constraints) that had not been proposed by other approaches before.

CONCLUSION

The international community is nowadays very interested in the use of constraints programming to support PL Engineering. Indeed, the analogy between PL specification formalisms and CP can easily be drawn: both are collections of variables and constraints that should be satisfied. We believe, like many other researchers in the community, that specifying a product line as a constraint program rather than with a more traditional formalism such as a feature model (Kang et al. 1990) 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 just boolean values as in Feature-Oriented Domain Analysis (FODA) models (Kang et al. 1990). 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 directly be executed and analyzed by off-the-shelf solvers. This last 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.

This paper showed how 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

constraints that apply to integer variables and other constraints too. 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

Akers, S. (1978). Binary Decision Diagrams. *IEEE Transactions on Computers, C*-27(6), 509–516.

Apt, K.R., Wallace M.G. (2006). *Constraint Logic Programming Using ECLiPSe*. Cambridge, Cambridge University Press.

Batory, D. (2005). Feature models, grammars, and propositional formulas. In *Software Product Lines Conference: Volume 3714 of Lecture Notes in Computer Sciences* (pages 7–17). Springer–Verlag.

Benavides, D. (2007). On *the Automated Analysis of Software Product Lines Using Feature Models. A Framework for Developing Automated Tool Support.* Unpublished doctoral dissertation, University of Seville, Spain.

Benavides, D., Ruiz-Cortés, A., & Trinidad, P. (2005). Using constraint programming to reason on feature models. In the *Seventeenth International Conference on Software Engineering and Knowledge Engineering*, pages 677–682.

Benavides, D., Segura, S., Trinidad, P., & Ruiz-Cortés, A. (2006). Using Java CSP solvers in the automated analyses of feature models. In Post-Proceedings of the *Summer School on Generative and Transformational Techniques in Software Engineering*. PA: Springer LNCS 4143.
Benavides, D., Segura, S. & Ruiz-Cortés, A. (2010). Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Journal of Information Systems*. Elsevier.

Boucher, Q., Classen, A., Faber, P., & Heymans, P. (2010). Introducing TVL, a Text-based Feature Modelling. In the *International Workshop on Variability Modelling of Software-intensive Systems*. Linz, Austria.

Classen, A., Boucher, Q., & Heymans. (2011). A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer* Programming. 76(12), 1130-1143.

Clements, P., & Northrop, L. (2001). *Software Product Lines : Practices and Patterns*. Reading, MA, USA, Addison Wesley.

Collavizza, H., &Rueher, M. (2006). Exploration of the capabilities of constraint programming for software verification. International Conference on. Tools And Algorithms For The Construction And Analysis Of Systems (pp. 182–196). Vienna, Austria.

Colmerauer, A. (1990). An Introduction to Prolog III. *Communications of the ACM, vol. 33(7)*.

Czarnecki, K., Helsen, S., & Eisenecker, U.W. (2005). Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice, 10*(1), 7–29.

Czarnecki, K., Helsen, S., & Eisenecker, U. (2005). Staged configuration through specialization and multi-level configuration of feature models. *Software Process Improvement and Practice, 10*(2).

Djebbi, O., & Salinesi, C. (2007). RED-PL, a Method for Deriving Product Requirements from a Product Line Requirements Model. *International Conference on Advances in Information Systems Engineering, International Conference on Advanced Information Systems Engineering*. Trondheim, Norway. PA: Springer LNCS 4495.

Djebbi, O., Salinesi, C., & Diaz, D. (2007). Deriving Product Line Requirements: the RED-PL Guidance Approach. *Asian Pacific Software Engineering Conference*. Nagoya, Japan. PA: IEEE Computer Society Digital Library.

Djebbi, O., Salinesi, C., & Rolland, C. (2008). Product Line Requirements Configuration in the Context of Multiple Models, *INSIGHT, INCOSE, 11:3*, 19-20.

Djebbi, O., & Salinesi C. (2008). Towards an Automatic PL Requirements Configuration through Constraints Reasoning. *International Workshop on Variability Modelling of Software-intensive Systems*. Essen, Germany. PA: University of Duisburg-Essen Press.

Diaz, D. & Codognet, P. (2001). Design and Implementation of the GNU Prolog System. *Journal of Functional and Logic Programming, Vol. 2001, No. 6*.

Ecole de Mines de Nantes. (2010). *CHOCO solver*. Retrieved March 01, 2010, from http://www.emn.fr/z-info/choco-solver/index.html.

Jaffar, J., Michaylov, S., Stuckey, P. & Yap, R. (1992). The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems, 14(3)*.

Fleurey, F., Baudry, B., France, R.B., & Ghosh, S. (2007). A generic approach for automatic model composition. In *Holger Giese (Ed.): Models in Software Engineering, Workshops and Symposia at MoDELS 2007*, Springer (pp. 7–15). Nashville, TN, USA.

Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., & Goedicke M. (1992). Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering 2(1),* 31–57.

Jaffar, J., & Maher, M. J. (1994). Constraint logic programming: A survey. *Journal of Logic Programming, Vol. 19/20,* 503-581.

Kang, K., Cohen, S., Hess, J., Novak, W., & Peterson, S. (1990). *Feature- Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Carnegie Mellon University, Software Engineering Institute.

Karataş, A., Oğuztüzün, H., & Doğru, A. (2010). Mapping Extended Feature Models to Constraint Logic Programming over Finite Domains. *Software Product Line Conference*. Jeju Island, Korea. PA: Lecture Notes in Computer Science.

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).

Le Berre, D. (2010). SAT4J solver. Retrieved March 01, 2010, from www.sat4j.org.

Lora-Michiels, A., Salinesi, C., & Mazo, R. (2010). A Method based on Association Rules to Construct Product Line Model. *4th International Workshop on Variability Modelling of Software-intensive Systems*. Linz, Austria. PA: University of Duisburg-Essen Press.

Mannion, M. Using first-order logic for product line model validation. In Proceedings of the Second *Software Product Line Conference* (pp. 176–187), San Diego, CA, USA. PA: Springer LNCS 2379.

Mazo R. A Generic Approach for Automated Verification of Product Line Models. Ph.D. Thesis, Université Paris 1 Panthéon - Sorbonne. Paris France, 24 November 2011.

Mazo, R., Grünbacher, P., Heider, W., Rabiser, R., Salinesi, C., & Diaz, D. (2011)Using Constraint Programming to Verify DOPLER Variability Models. Proceedings of the *Workshop on Variability Modelling of Software-intensive Systems*. Namur, Belgium. PA: *ACM Press*.

Mazo, R., Lopez-Herrejon, R., Salinesi, C., Diaz, D., & Egyed, A. (2011). A Constraint Programming Approach for Checking Conformance in Feature Models. In *35th Annual International Computer Software and Applications Conference COMPSAC*. Munich, Germany. PA: IEEE Press.

Mazo, R., Salinesi, C., & Diaz, D. Abstract Constraints: A General Framework for Solver-Independent Reasoning on Product Line Models. (2011). Accepted on *INSIGHT - Journal of International Council on Systems Engineering (INCOSE),* to be released the 15 October 2011.

Mazo, R., Salinesi, C., Diaz, D., Lora-Michiels, A. (2011). Transforming Attribute and Clone-Enabled Feature Models Into Constraint Programs Over Finite Domains. *6th International Conference on Evaluation of Novel Approaches to Software Engineering*, Beijing, China. PA: Springer Press.

Mazo R., Salinesi C., Diaz D. VariaMos: a Tool for Product Line Driven Systems Engineering with a Constraint Based Approach. *24th International Conference on Advanced Information Systems Engineering (CAiSE Forum'12)*, Springer Press, Gdansk-Poland, 25 – 29 June 2012. https://sites.google.com/site/raulmazo/

Mendonça, M., Wasowski, A., & Czarnecki, K. (2009). SAT–based analysis of feature models is easy. In Proceedings of the *Sofware Product Line Conference*. San Francisco, California, USA. PA: ACM Digital Library.

Pohl K., Bockle,G., van der Linden,F.J. Software Product Line Engineering: Foundations, Principles and Techniques.Springer (2005)

Salinesi, C., Mazo, R., & Diaz, D. (2010). Criteria for the verification of feature models. In *28th INFORSID Conference*, (pp. 80-96). Marseille, France.

Salinesi, C., Mazo, R., Diaz, D., & Djebbi, O. (2010). Solving Integer Constraint in Reuse Based Requirements Engineering. 18th *IEEE International Conference on Requirements Engineering*. Sydney, Australia. PA: IEEE Press.

Salinesi, C., Mazo, R., Djebbi, O., Diaz, D., Lora-Michiels, A. (2011). Constraints: the Core of Product Line Engineering. *Fifth IEEE International Conference on Research Challenges in Information Science* (pp. 29-38). Guadeloupe-French West Indies, France. PA: IEEE Press.

Salinesi, C., Rolland, C., Mazo, R. (2009). VMWare: Tool Support for Automatic Verification of Structural and Semantic Correctness in Product Line Models. In *International Workshop on Variability Modelling of Software-intensive Systems* (pp. 85-90). Sevilla, Spain. PA: University of Duisburg-Essen Press.

Saraswat, V. (1992). The Category of Constraint Systems is Cartesian-Closed. *Logic In Computer Science* (pp. 341-345, 1992), IEEE Press.

S. de Boer, F., & Palamidessi, C. (1991). A Fully Abstract Model for Concurrent Constraint Programming. *TAPSOFT, Vol.1*, (pp. 296-319). TAPSOFT Editors.

Segura, S., Benavides, D., & Ruiz-Cortés, A., & Trinidad, P. (2008). Automated merging of feature models using graph transformations. In *international summer schools on Generative and Transformational Techniques in Software Engineering Volume 5235*. Springer-Verlag LNCS.

Specification and Verification Center at CM University. (2010). SMV system. Retrieved March 01, 2010, from www.cs.cmu.edu/~modelcheck

Trinidad, P., Benavides, D., Duran, A., Ruiz-Cortés, A., & Toro, M. (2008). Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software, 81*(6), 883–896.

White, J., Doughtery, B., Schmidt, D., & Benavides, D. (2009). Automated reasoning for multi-step software product line configuration problems. In *Proceedings of the Software Product Line Conference,* (pp. 11–20). San Francisco, CA. USA. PA: ACM Digital Library.

White, J., Schmidt, D., Benavides, D., Trinidad, P., & Ruiz-Cortes A. (2008). Automated diagnosis of product-line configuration errors in feature models. In Proceedings of the *Software Product Line Conference*. Limerick, Ireland. PA: ACM Digital Library.

Van der Maßen, T., & Lichter, H. (2002). Modeling Variability by UML Use Case Diagrams. International Workshop on Requirements Engineering for Product Lines. *Avaya Labs Report series editors,* (pp. 18-26). Essen, Germany.

Zhang,W., Zhao, H., & Mei, H. (2004). A propositional logic-based method for verification of feature models. In *Jim Davies, Wolfram Schulte, Michael Barnett (Eds.), Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods volume 3308*, (pp. 115–130). Seattle, WA, USA, Springer–Verlag.

Ziadi, T. (2004). *Manipulation de Lignes de Produits en UML*. Unpublished doctoral dissertation, IRISA-TRISKELL, Université de Rennes 1, France.

Raúl Mazo is an Adjunct Assistant Lecturer at the Panthéon Sorbonne University and at the Engineering School EFREI, both located in Paris, France. Before joining the Panthéon Sorbonne University, Raúl Mazo worked as a Telecommunications Engineer at the University of Antioquia, Colombia (2004-2006). He received a Computer Science Engineering degree in 2005 from the University of Antioquia, Colombia, and a Master of Science degree in Information Systems in 2008 from the Panthéon Sorbonne University. At present, he is a Ph.D. student at the Panthéon Sorbonne University and he intends to finish the Ph.D. in November 2011.
His research interests include software design modeling, requirements engineering, product lines engineering, ERP systems and network security. His research work focuses on three topics of product line engineering: (i) definition of constraint-based formalisms to represent complex product line models; (ii) construction of product line models; and (iii) definition of methods to validate, verify and analyse variability models.

Dr Camille Salinesi, Professor at Université Paris 1 Panthéon - Sorbonne is the head of the Centre de Recherche en Informatique, specialized in Information Systems Engineering. He published more than 70 refereed papers in international conferences and scientific journals on various topics such as requirements engineering, strategic alignment, or product lines.
Prof. Salinesi was involved in fundamental research projects (FP4 NATURE, FP5 CREWS) and was the leader for collaborations and consultancy works for various companies such as France Télécom, SNCF, Renault, MédiaScience, and EDF) He chaired the REFSQ, REP, and RIGIM workshops and in 2005, he was Organizing Chair of the 13th IEEE International Conference on Requirements Engineering.

Prof. Salinesi teaches various topics such as Databases, Requirements Engineering, Enterprise Resource Planning, Information Systems Management, and Enterprise Architecture.
Prof. Salinesi is member of IEEE and INCOSE.

Daniel Diaz is an Assistant Professor at the University of Paris 1 - Pantheon Sorbonne (FRANCE). Hi is Member of the CRI (Reaserch Center on Computing Science). Daniel Diaz is the author of GNU Prolog. Previsouly, hi collaborated with the laboratoire INRIA center Rocquencourt (FRANCE) in the Contraintes Project. His Research Topics are: Logic Programming, Constraint Programming, Local search and parallelism.

After she was graduated from a Master in Distributed Computing at University of Paris 6, Dr. Olfa Djebbi conducted her doctoral research within the company Diagnostica Stago, with a fund support by the French government.
Dr. Olfa Djebbi is interested in Software Engineering, especially in requirements engineering and product lines development.
Since 2005, she has been involved in the works of the French association of systems engineers AFIS, French branch of the INCOSE (International Council on Systems Engineering), and which brings together industrialists and academics around issues of current interest.
Moreover, Mrs. Djebbi worked as teacher and researcher at Sorbonne and Dauphine Universities and has organized and led missions in various companies (including Peugeot, Renault, ADN and Sanofi Pasteur).
She has numerous publications in class-A conferences including the IEEE International Requirements Engineering Conference and the International Conference on Advanced Information Systems Engineering, as well as more industry-oriented journals such as INCOSE's INSIGHT.

Alberto Lora Michiels: Industrial Engineer 1999 (Escuela Colombiana de Ingenieria), Certificate in Statistics (Universidad Nacional de Colombia 2005), M2 en Système d'information et décision (Univeristé Paris 1 Panthéon Sorbonne). Currently, Material Requirement Planner Leader Supply Chain Baxter International Inc Lessines Belgique. Research interests: Product line model construction and data mining.

Anexe

Tables 2 to 6 compile the constructs, and its corresponding CP representations, of the most popular languages used to specify PLMs.

Table 2. Compilation of the feature-based languages' constructs and the corresponding representation as CPs.

Constructor and domains vs. Languages	**FODA-like models** (Kang et al., 2002)	**Feature models with cardinalities** (Riebisch et al. 2002; Czarnecki et al. 2005) **and attributes** (Benavides et al. 2005c; Streitferdt et al. 2006; White et al. 2009)
Root. The root element must be selected in all the configurations.	If {Root} ∈ {true, false} then Root = true If {Root} ∈ {0, 1} then Root = 1	If {Root} ∈ {true, false} then Root = true If {Root} ∈ ℤ then Root ≥ 1

Optional. If the father element is selected, the child element can but needs not be selected. Otherwise, if the child element is selected, the father element must as well be selected.	If {Father, Child} ∈ {true, false} then Child ⇒ Father If {Father, Child} ∈ {0, 1} then Father ≥ Child	If {Father, Child} ∈ {true, false} then Child ⇒ Father If {Father, Child} ∈ {0, 1} then Father ≥ Child If {Father, Child} ∈ ℤ then Child ≥ 1 ⇒ Father ≥ 1
Mandatory. If the father element is selected, the child element must be selected as well and vice versa.	If {Father, Child} ∈ {true, false} then Father ⇔ Child If {Father, Child} ∈ {0, 1} then Father = Child	If {Father, Child} ∈ {true, false} then Father ⇔ Child If {Father, Child} ∈ {0, 1} then Father = Child If {Father, Child} ∈ ℤ then Child ≥ 1 ⇔ Father ≥ 1
Requires (includes). If the requiring element is selected, the required element(s) has(have) to be selected as well, but not vice-versa.	If {Requiring, Required} ∈ {true, false, 0, 1} then Requiring ⇒ Required	If {Requiring, Required} ∈ {true, false, 0, 1} then Requiring ⇒ Required If {Requiring, Required} ∈ ℤ then Requiring ≥ 1 ⇒ Required ≥ 1
Exclusion. Indicates that both excluded elements cannot be selected in one product configuration.	If {Excluding, Excluded} ∈ {true, false} then Excluding ⊕ Excluded If {Father, Child} ∈ {0, 1} then Excluding + Excluded ≤ 1	If {Excluding, Excluded} ∈ {true, false} then Excluding ⊕ Excluded If {Father, Child} ∈ {0, 1} then Excluding + Excluded ≤ 1 If {Father, Child} ∈ ℤ then Excluding * Excluded = 0
Alternative/xor-decomposition. A set of child elements are defined as alternative if only one element can be selected when its parent element is part of the product.	If {Father, Child1, …, ChildN} ∈ {true, false} then: (Child1⇔(¬Child2 ⋀...⋀ ¬ChildN ⋀ Father) ⋀ Child2 ⇔ (¬Child1 ⋀...⋀ ¬ChildN ⋀ Father) ⋀ ChildN ⇔ (¬Child1 ⋀...⋀ ¬ChildN-1 ⋀ Father))	
Or-Relation. A set of child elements are defined as an or-relation if one or more of them can be included in the products in which its parent element appears.	If {Father, Child1, …, ChildN} ∈ {true, false} then: Father ⇔ Child1 ∨...∨ ChildN	
Group cardinality. Cardinality determines how many variants (with the same father) may be chosen, at least M and at most N of the group. Besides, if one of the children is selected, the father element must be selected as well.		If {Father, Child1, …, ChildN} ∈ {0, 1} then Father ≥ Child1 ⋀ …⋀ Father ≥ ChildN ⋀ M*Father ≤ Child1+...+ChildN ≤ N*Father
A **feature cardinality** is represented as a sequence of intervals [min..max] determining the number of instances of a particular feature that can be part of a product. Each instance is called a clone.		If {Father, Clone1, …, CloneN} ∈ { 0, 1} then: Clone1 ⇒ Father ⋀ ...⋀ CloneN ⇒ Father ⋀ Father ⇒ (M ≤ Clone1 + … + CloneN ≤ N)
Attribute. An attribute is a variable associated to a reusable element. An attribute has a name, a domain, and a value (consistent with the domain) at a given configuration time.		value ∈ Domain ⋀ Attribute = value ⋀ ReusableElement⇔Attribute > 0
Domain of the variables found in the language	Boolean.	Features are usually boolean, but they can also be Integer or Real variables. However if there are attributes in the FODA model, they are usually Integers or Reals
Constraints type: boolean, linear,	Boolean, reified, =, >, >, ≥, ≤	Boolean, arithmetic, polynomial,

polynomial, reified, symbolic, others. Operators used.		symbolic, reified, =, >, >, ≥, ≤
Special operators: Different and negation	¬	≠

Table 3. Compilation of the OVM and TVL's constructs and the corresponding representation as CPs.

Constructor and domains vs. Languages	**Orthogonal Variability Models (OVM)** (Pohl et al. 2005)	**Textual Variability Language (TVL)** (Boucher et al. 2010)
Root. The root element must be selected in all the configurations.		root Element
Dependency/and-decomposition: operator allOf. The selection of the children depends of the selection of the father element and vice versa		Father ⇔ (Child1 ⋀…⋀ ChildN)
Optional. If the father element is selected, the child element can but needs not be selected. Otherwise, if the child element is selected, the father element must as well be selected.	If {Element1, Element2} ∈ {true, false} then Element2 ⇒ Element1 if {Element1, Element2} ∈ {0, 1} then Element1 ≥ Element2 If {Element1, Element2} ∈ ℤ then Element2 ≥ 1 ⇒ Element1 ≥ 1	Child ⇒ Father
Mandatory. If the father element is selected, the child element must be selected as well and vice versa.	If {Element1, Element2} ∈ {true, false} then Element1 ⇔ Element2 If {Element1t, Element2} ∈ {0, 1} then Element1 = Element2 if {Element1, Element2} ∈ ℤ then Element2 ≥ 1 ⇔ Element1 ≥ 1	Father ⇔ Child
Requires (includes). If the requiring element is selected, the required element(s) has(have) to be selected as well, but not vice-versa.	If {Requiring, Required} ∈ {true, false, 0, 1} then Requiring ⇒ Required If {Requiring, Required} ∈ ℤ then Requiring ≥ 1 ⇒ Required ≥ 1	If {Requiring, Required} ∈ {true, false, 0, 1} then Requiring ⇒ Required
Exclusion. Indicates that both excluded elements cannot be selected in one product configuration.	If {Excluding, Excluded} ∈ {true, false} then Excluding ⊕ Excluded If {Father, Child} ∈ {0, 1} then Excluding+Excluded ≤ 1 If {Father, Child} ∈ ℤ then Excluding * Excluded = 0	
Alternative/xor-decomposition. A set of child elements are defined as alternative if only one element can be selected when its parent element is part of the product.		If {Father, Child1, …, ChildN} ∈ {0, 1} then: (Child1⇔(¬Child2 ⋀...⋀ ¬ChildN ⋀ Father) ⋀ Child2 ⇔ (¬Child1 ⋀...⋀ ¬ChildN ⋀ Father) ⋀ ChildN ⇔ (¬Child1 ⋀...⋀ ¬ChildN-1 ⋀ Father))
Or-Relation. A set of child elements are defined as an or-relation if one or more of them can be included in the products in which its parent element appears.		If {Father, Child1, …, ChildN} ∈ {0, 1} then: Father ⇔ Child1 ⋁...⋁ ChildN
Group cardinality. Cardinality determines how many variants (with the same father) may be chosen, at least M and at most N of the group. Besides, if one of the children is selected, the father	If {VariationPoint, Variant1, …, VariantN} ∈ {0, 1} then VariationPoint ≥ Variant1 ⋀ …⋀ VariationPoint ≥ VariantN ⋀ M*VariationPoint ≤ Variant1+...+ VariantN ≤ N*VariationPoint	If {Father, Child1, …, ChildN} ∈ {0, 1} then Father ≥ Child1 ⋀ …⋀ Father ≥ ChildN ⋀ M*Father ≤ Child1+...+ChildN ≤ N*Father

element must be selected as well.		
Individual cardinality is represented as a sequence of intervals [min..max] determining the number of instances of a particular feature that can be part of a product. Each instance is called a clone.	If {Father, Clone1, …, CloneN} ∈ { 0, 1} then: Clone1 ⇒ Father ⋀ ...⋀ CloneN ⇒ Father ⋀ Father ⇒ (M ≤ Clone1 + … + CloneN ≤ N)	If {Father, Clone1, …, CloneN} ∈ { 0, 1} then: Clone1 ⇒ Father ⋀ ...⋀ CloneN ⇒ Father ⋀ Father ⇒ (M ≤ Clone1 + … + CloneN ≤ N)
Attribute. An attribute is a variable associated to a reusable element. An attribute has a name, a domain, and a value (consistent with the domain) at a given configuration time.		Attribute ∈ { integer, real, boolean, enumeration} ⋀ Attribute = value ⋀ ReusableElement ⇔ Attribute > 0
Domain of the variables found in the language	Boolean, Integer, Real	Features: boolean. Attributes: integer (int), real (real), Boolean (bool) and enumeration (enum)
Constraints type: boolean, linear, polynomial, reified, symbolic, others. Operators used.	Boolean, arithmetic, polynomial, symbolic, reified, =, >, >, ≥, ≤	All the constraints of C: +, -, /, *, abs, for numeric values; !, &&, jj, ->, <-> for Boolean values as well as comparison operators >, >=, < or <=. And aggregation functions like sum, mul, min, max, avg, count, and, or and xor
Special operators: Different and negation	≠	≠, ¬

Table 4. Compilation of the Class-based and Use case-based variability languages' constructs and the corresponding representation as CPs.

Constructor and domains vs. Languages	**Class-based PLMs** (Ziadi 2004), (Korherr, List 2007)	**Use case-based PLMs** (Van der Maßen, Lichter 2002)
Optional. If the father element is selected, the child element can but needs not be selected. Otherwise, if the child element is selected, the father element must as well be selected.	If {Element1, Element2} ∈ {true, false} then Element2 ⇒ Element1 If {Element1, Element2} ∈ {0, 1} then Element1 ≥ Element2	If {Element1, Element2} ∈ {true, false} then Element2 ⇒ Element1 If {Element1, Element2} ∈ {0, 1} then Element1 ≥ Element2
Mandatory. If the father element is selected, the child element must be selected as well and vice versa.	If {Element1, Element2} ∈ {true, false} then Element1 ⇔ Element2 If {Element1t, Element2} ∈ {0, 1} then Element1 = Element2	If {Element1, Element2} ∈ {true, false} then Element1 ⇔ Element2 If {Element1t, Element2} ∈ {0, 1} then Element1 = Element2
Requires (includes). If the requiring element is selected, the required element(s) has(have) to be selected as well, but not vice-versa.	If {Requiring, Required} ∈ {true, false, 0, 1} then Requiring ⇒ Required If {Requiring, Required} ∈ ℤ then Requiring ≥ 1 ⇒ Required ≥ 1	If {Requiring, Required} ∈ {true, false, 0, 1} then Requiring ⇒ Required If {Requiring, Required} ∈ ℤ then Requiring ≥ 1 ⇒ Required ≥ 1
Group cardinality. Cardinality determines how many variants (with the same father) may be chosen, at least M and at most N of the group. Besides, if one of the children is selected, the father element must be selected as well.		If {Father, Child1, …, ChildN} ∈ {0, 1} then Father ≥ Child1 ⋀ …⋀ Father ≥ ChildN ⋀ M*Father ≤ Child1+...+ChildN ≤ N*Father
Individual cardinality is	If {FatherClass, Clone1, …, CloneN} ∈	

represented as a sequence of intervals [min..max] determining the number of instances of a particular feature that can be part of a product. Each instance is called a clone.	{ 0, 1} then: Clone1 ⇒ FatherClass ∧ ...∧ CloneN ⇒ FatherClass ∧ FatherClass ⇒ (M ≤ Clone1 + … + CloneN ≤ N)	
Domain of the variables found in the language	Boolean, Integer, Real	Boolean
Constraints type: boolean, linear, polynomial, reified, symbolic, others. Operators used.	Boolean, arithmetic	Boolean, arithmetic

Table 5. Compilation of the Dopler and CEA variability languages' constructs and the corresponding representation as CPs.

Constructor and domains vs. Languages	**Dopler variability language** (Dhungana et al. 2010)	**CEA - variability language**
Root/Visibility Condition. The root decision must be solved in all the configurations.	Decision = true ∨ Decision = false	
Mandatory. If the father element is selected, the child element must be selected as well and vice versa.		If {Element1, Element2} ∈ {true, false} then Element1 ⇔ Element2 If {Element1, Element2} ∈ {0, 1} then Element1 = Element2
Requires/Decision Effects/ Inclusion Conditions. If the requiring element is selected, the required element(s) has(have) to be selected as well, but not vice-versa.	Constraint1 ⇒ Constraint2; Asset ⇒ Decision	
Validity condition. RDL equivalent: "sauf". The Validity Condition constrains the range of possible values for a particular decision.		If {Element1, Element2} ∈ {true, false, 0, 1} then Element1 ⇒ Element2 If {Element1, Element2} ∈ ℤ then Element1 ≥ 1 ⇒ Element2 ≥ 1
Asset Dependencies define relationships between assets. Arbitrary relationship types with different semantics like requires, contributes to, excludes or implements.		If {Element1, Element2, …, ElementN} ∈ {true, false} then: Element1 ∨... ∨ ElementN = true If {Element1, Element2, …, ElementN} ∈ {1, 0} then: Element1 + ...+ElementN ≥ 1
Group cardinality/ Enumeration Decision Type/. Cardinality determines how many variants (with the same father) may be chosen, at least M and at most N of the group. Besides, if one of the children is selected, the father element must be selected as well.	Decision ∈ ValidityCondition ∧ Decision ≥ DecisionOption1 ∧ …∧ Decision ≥ DecisionOptionN ∧ M*Decision ≤ DecisionOption1+...+ DecisionOptionN ≤ N*Decision	
Domain of the variables found in the language	Boolean, String, Number(real) and Enumeration	
Constraints type: boolean, linear, polynomial, reified, symbolic, others. Operators used.	All the constraints of Java	
Special operators: Different and negation	≠, ¬	

Table 6. Compilation of the RDL and Latice variability languages' constructs and the corresponding representation as CPs.

Constructor and domains vs. Languages	**Renaul Documentary Language (RDL)**	**Latice** (Mannion 2002)
Root. The root element must be selected in all the configurations.	root Projet_Vehicule	
Dependency/and-decomposition: operator allOf. The selection of the children depends of the selection of the father element and vice versa		Father ⋀ (Child1 ⋀…⋀ ChildN)
Optional. If the father element is selected, the child element can but needs not be selected. Otherwise, if the child element is selected, the father element must as well be selected.	if {Use_Case, Element} ∈ {true, false} then Element ⇒ Use_Case if {Use_Case, Element} ∈ {0, 1} then Use_Case ≥ Element	
Mandatory. If the father element is selected, the child element must be selected as well and vice versa.	if {Use_Case, Element} ∈ {true, false} then Use_Case ⇔ Element if {Use_Case, Element} ∈ {0, 1} then Use_Case = Element	Father ⇔ Child
Requires (includes). If the requiring element is selected, the required element(s) has(have) to be selected as well, but not vice-versa.	if {Requiring, Required} ∈ {true, false, 0, 1} then Requiring ⇒ Required If {Requiring, Required} ∈ ℤ then Requiring ≥ 1 ⇒ Required ≥ 1	
Exclusion. Indicates that both excluded elements cannot be selected in one product configuration.	if {Excluding, Excluded} ∈ {true, false} then Excluding ⇒ ¬Excluded if {Excluding, Excluding} ∈ {0, 1} then Excluding - Excluded ≥ 1	Excluding ⊕ Excluded
Alternative/xor-decomposition. A set of child elements are defined as alternative if only one element can be selected when its parent element is part of the product.	if {Use_Case, Element1, …, ElementN} ∈ {true, false} then: (Element1⇔(¬Element2 ⋀...⋀ ¬ElementN ⋀ Father) ⋀ Element2 ⇔ (¬Element1 ⋀...⋀ ¬ElementN ⋀ Use_Case) ⋀ ElementN ⇔ (¬Element1 ⋀...⋀ ¬ElementN-1 ⋀ Use_Case)) if {Use_Case, Element1, …, ElementN} ∈ℤ then: Use_Case - (Element1 + ...+ ElementN) = 0	
Or-Relation. A set of child elements are defined as an or-relation if one or more of them can be included in the products in which its parent element appears.	if {Father, Child1, …, ChildN} ∈ {true, false} then: Father ⇔ Child1 ∨...∨ ChildN if {Use_Case, Element1, …, ElementN} ∈ℤ then: Use_Case - (Element1 + ...+ ElementN) ≥ 0	Father ⇔ Child1 ∨...∨ ChildN
Validity condition. RDL equivalent: "sauf". It constrains the range of possible values for a particular use case.	if {Relation1, Use_Case} ∈ {true, false} then Relation1 ⇒ Use_Case if {Relation1, Use_Case} ∈ {0, 1} then Use_Case - Relation1 ≥ 0	
Conjunction of subgraphs. If Gi and Gj are the logical expressions for two different subgraphs of a lattice, the PLM is con conjunction of Gi and Gj		Gi ⋀ Gj
Domain des variables du language.		Boolean
Constraints type: boolean, linear, polynomial, reified, symbolic, others. Operators used.		Boolean