Analysis of Constraints including Clonable Features using Hydra

Pablo Sánchez
Dpto. Matemáticas, Estadística y Computación
Universidad de Cantabria
Santander (Cantabria, Spain)
p.sanchez@unican.es

José Ramón Salazar, Lidia Fuentes
Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga
Málaga (Málaga, Spain)
{*salazar, lff*}*@lcc.uma.es*

***Abstract*—**

Clonable features, i.e. features whose cardinality has an upper bound greater than 1, were proposed several years ago. Although they have been included in several feature modelling tools, there is still no tool able to properly deal with constraints, such as dependencies or mutual exclusions, which include clonable features. The first challenge these tools find is that the semantics of these constraints becomes undefined in the presence of clonable features. As a consequence of this lack of semantics, the analysis and validation of these constraints is simply not feasible. To overcome these limitations, this paper presents: (1) a language for specifying external constraints involving clonable features with a clearly defined semantics; and (2) how to analyse these constraints by transforming them into a Constraint Satisfaction Problem (CSP). This language and the analyser have been incorporated into our feature modelling tool, called *Hydra*. We validate our ideas by applying them to a SmartHome industrial software product line.

***Keywords*-Feature Models, Clonable Features, Constraint Analysis**

I. Introduction

Clonable features in features models are features with an multiplicity upper bound greater than 1 [4], [5]. They are used to model features that can appear with a variable number of instances in the different products belonging to a same Software Product Line (SPL). Clonable features were introduced in Czarnecki et al [4], almost a decade ago, mainly due to practical and industrial reasons [4]. In such a work, clonable features were used to model a SPL for implementing a standard of the European Space Agency for satellite communication. In this SPL, a software application might have several communication interfaces; and each service had to be individually configurable. Therefore, communication interface was included in the feature model as a clonable feature with several subfeatures. Since they were created, clonable features have been incorporated to several feature modelling tools, such as FMP[1], Hydra[2], Moskitt Feature Modeller[3] or xFeature[4].

As the experienced reader probably already knows, parent-child relationships between features are not often enough to capture all the relationships between features, being required the use of *external* or *cross-tree constraints* [1]. Typical examples of these cross-tree constraints are dependencies and mutual exclusions. Despite of clonable features were introduced almost ten years ago, there is no tool properly supporting the expression and analysis of cross-tree constraints yet. This shortcoming is mainly because of the semantics of these cross-tree constraints become unspecified when they include clonable features. Therefore, the analysis of these constraints is simply impossible due to the lack of a clearly defined semantics. Moreover, the expressiveness of the languages used to specify these constraints is often not enough. As a consequence, cross-tree constraints including clonable features are neither specified nor included in the feature model, which implies users and stakeholder might create configurations violating these constraints, which would lead to erroneous products. Even in the case when these constraints are specified somehow, due to the lack of tool-support, they must be analysed and validated manually, which is a tedious, cumbersome and error-prone task.

To overcome these shortcomings, this paper presents: (1) a language for specifying constraints including clonable features with a clearly defined semantics; and (2) how to analyse these constraints by transforming them into a Constraint Satisfaction Problem (CSP) [10]. Using these language and this transformation process, it is possible to specify expressive constraints including clonable features and automatically analyse them using third-party libraries for CSP solving.

This language has been incorporated into our feature modelling tool, called *Hydra*. We have also implemented several analysis operations [2] by transforming these constraints into a CSP problem, which is solved using a third-party library for CSP solving called Choco [8]. More specifically, we have implemented TO BE COMPLETED.

This work has been supported by the Spanish Ministry Project TIN2008-01942/TIN, the EC STREP Project AMPLE IST-033710 and the Junta de Andalucía regional project FamWare TIC-5231

[1]http://gsd.uwaterloo.ca/fmp

[2]http://caosd.lcc.uma.es/spl/hydra/

[3]http://www.pros.upv.es/mfm/

[4]http://www.pnp-software.com/XFeature/

Using Hydra, the new language and the constraint analyser, we have modelled, configured and validated feature models for: (1) a SmartHome software product line based on an industrial case study [7]; (2) a graphical user interface, based on a domain specific language, presented in Santos et al [9]; a (3) the feature model for the satellite communication systems which initially motivated the addition of clonable features to feature models.

TO BE UPDATED AFTER FINISHING THE PAPER

After this introduction, this paper is structured as follows: Section II provides some background on clonable features and analyse the research challenges they create. Section IV presents the new language for specifying constraints and how to analyse them. Section VI comments on related work. Section VII concludes the paper, provides a critical reflection on the benefits of our approach. and outlines future work.

II. Motivation

This section explains the shortcomings of feature modelling tools we faced during the development of a SmartHome SPL in the context of the AMPLE project. We comment on the research challenges we discovered and that we aim to solve in this paper.

A. A feature model for a Smart Home SPL

Figure 1 shows the feature model developed in the context of the AMPLE project for modelling a Smart Home SPL, based on a industrial case study provided by Siemens AG [7][5]. Clonable features were used in this case study to specify that an automated house can have several floors, with several rooms per floor. Each room can have a different number of devices, such as lights, heaters or windows, which can be automated.

The SmartHome SPL offers a set of services that can be included in the final software for a specific house. These services controls automatically a set of devices, such as windows, heaters or lights. Moreover, some advanced functions, such as Presence Simulation or Smart Energy Management, can be selected. This latter feature is on charge of coordinating windows and heaters to save energy. For instance, before heating a room, this feature will automatically close all the windows in that room to avoid wasting energy.

We would like to comment the feature model might be simplified by using *feature references* [6]. Feature references are often introduced in feature models to improve scalability by means of avoiding redundancies. We might have created a feature model called Facilities and to add LightMng, WindowMng, HeaterMng and SmartEnergyMng as subfeatures. Then, GeneralFacilities, FloorFacilities and RoomFacilities would simply refer to Facilities, avoiding replications. Although Hydra, our feature modelling tool, supports feature references, we have not used for this paper for the sake of simplicity, because they introduces some extra complexity in the explanations of the constraint analysis process which is not relevant for the purpose of this paper.

Figure 2 shows an example of configuration, or *specialization* [5] of the feature model of Figure 1.

The process of creating new instances of a clonable feature is called *cloning*, and its semantics was clearly defined in Czarnecki et al [5]. This semantics specifies each time a new instance of a clonable feature is created, the subtree below the clonable feature is copied below this newly created feature. This subtree can then be configured as any other feature model. Changes in the subtree below an instance of a clonable feature have no influence on subtrees belonging to other instances. Therefore, each instance of a clonable feature can have its own configuration. To distinguish between different instances, or clones, of a same feature, we have opted for giving a different name to each instance. For instance, each instance of the Floor feature has a different name (e.g. Ground, First), which indicates what floor is.

The configuration depicted in represents a simple house, with two floors - a ground floor and a first floor, and three rooms- a kitchen and a living room in the first floor and a bedroom in the first floor. Window Managament has been selected for the whole house; whereas LightMng is only selected for the Kitchen and the Bedroom. Heater and Smart Energy Management has been included in the living room. It should be noticed that the usage of clonable features allows a more fine-grained configuration of software products. Due to the use of clones for the different floors and rooms, we can customise each floor and/or room individually.

The reader could argue a Domain-Specific Language or Ecore-based model would be more useful for this case study than a feature model. We will skip this question at this moment and we will come back to it in the discussion section.

B. Research challenges on clonable features

Clonable features creates new challenges when the specification of cross-tree constraints is required. For instance, according to Figure 1, house facilities can be selected at a house, floor or room level. Therefore, a cross-tree constraint should specify if LightMng has been selected at the floor level, this facility must also have been selected for each room belonging to that floor.

Since SmartEnergyMng requires the HeaterMng and WindowMng features have been selected, other constraint should specify whenever SmartEnergyMng has been selected at one level (e.g. for a specific floor), HeaterMng and WindowMng have also been selected at the same level (i.e. for that specific floor). Finally, we might want to specify more advanced constraints, like if Presence Simulation is going to be included in a automated house, a 25% of the rooms in

[5]This feature model, as well as all the material presented throughout this paper, can be downloaded from TODO: URL TO BE PROVIDED

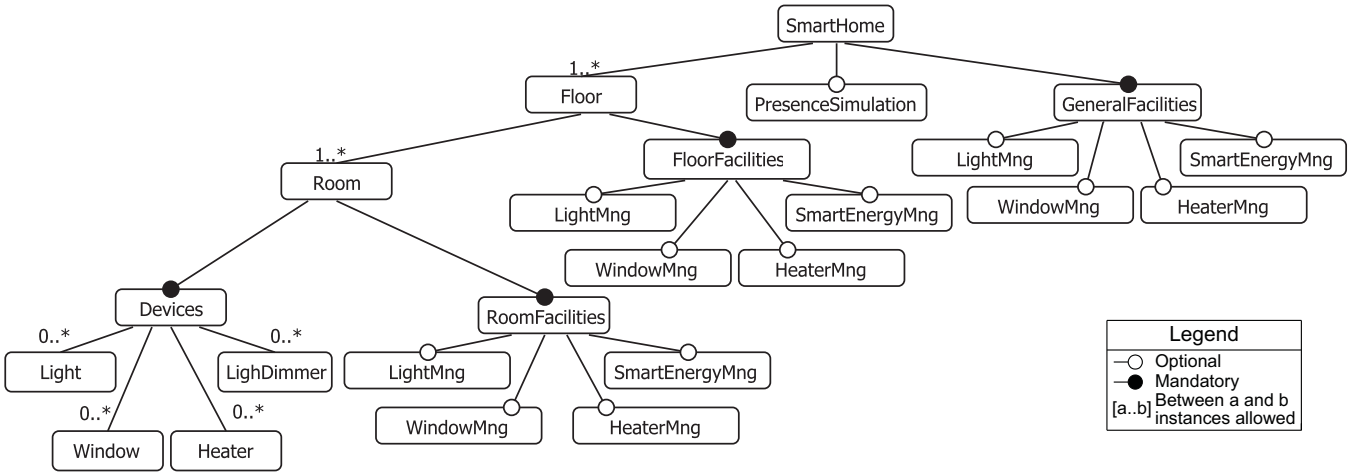

Figure 1. Feature model for a SmartHome SPL

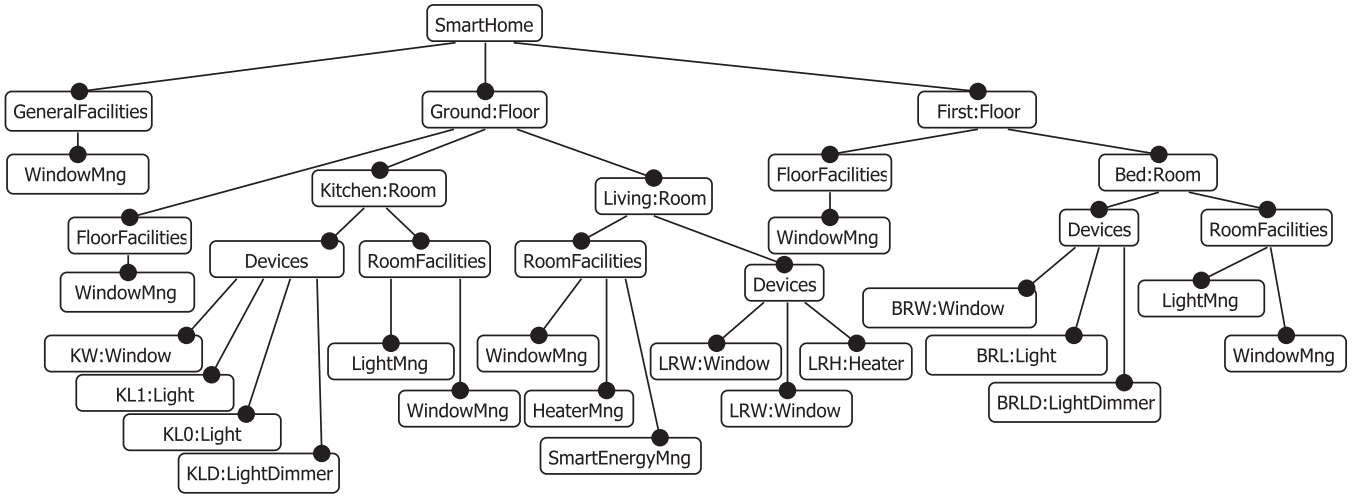

Figure 2. Configuration of a specific automated house

the house at least must include automatic light management .

In feature models, the most usual way to express cross-tree constraints is using propositional logic formulas, such as SmartEnergyMng implies (LightMng and HeaterMng), where the different features are the atoms for these formulas [1]. An atom is evaluated to true when the corresponding feature has been selected, otherwise it is false. The main problem when dealing with clonable features is that this correspondence cannot be used. Clonable features are not selected or unselected. Clonable are just cloned. So, the meaning of a atom related to a clonable feature becomes undefined. As a result, a propositional logical formula including clonable features can not be evaluated and it can not be determined when the corresponding constraint has been satisfied or violated.

For instance, let us suppose A and B are clonable features. In this case, what would be the meaning of a external constraint like A implies B? Does this means that one instance of A implies the existence of at least one instance of B? Or, on the other hand, does it means that the existence of all potential As implies the existence of all potential Bs? Or, why not, the existence of all potential As implies the existence of only one B? So, the first research challenge we face is to decide what a clonable feature exactly means in a logical formula expressing a cross-tree constraint.

Each clonable feature really evaluates to a set of clones. For instance, the feature Floor in Figure 1 actually evaluates to {Ground, First }, instead of selected or unselected. Thus, we might want to specify properties that only applies to: (1) at least one element in that set (e.g. only to the Ground

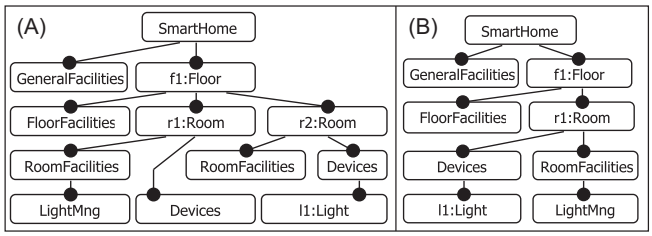

Figure 3. (Left) Invalid configuration (Right) Valid configuration

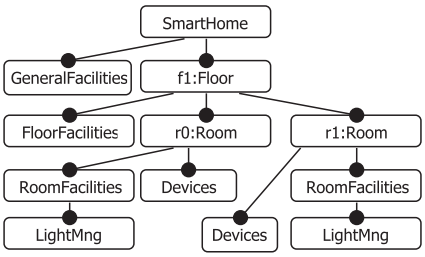

Figure 4. The *multiple features* phenomena

floor); (2) to all the elements in that set that fulfill a certain constraint (e.g to all the floors with LightMng selected in one room at least); or (3) all the elements in the set. This would allow us to express more complex constraints such as, such as: if HeaterMng is selected per house level, one Room at least must contain one Heater as a minimum. So, our second research challenge is to add quantification mechanisms to constraints involving clonable features.

Moreover, as already commented in previous section, we might want to specify constraints which must be evaluated for a particular subtree of the whole feature model, i.e. in a particular *context*. For instance, if LightMng has been selected for a particular Room, such a Room must have one Light at least. Thus, we should specify a constraint like "LightMng implies one Light device at least". If we do not limit the scope in which this constraint is evaluated, this constraint will be true for the configurations of Figure 3 (a) and (b). Nevertheless, this constraint should be false for Figure 3 (a), since r1:Room has selected the LightMng facility, but it has not any Light device to control. This means this constraint must be evaluated for all rooms (notice we are using again quantification) and using only the subtree below each Room. Thus, the third research challenge is how to specify and analyse constraints which must be must be evaluated in the scope of a particular context.

Finally, we would to like to point out that not only clonable features can have more than one instance per configuration of a feature model. All those features that have as an ancestor a clonable features can appear more than once in a configuration model. We will call to this kind of feature, i.e. non clonable features with a clonable ancestor, *multiple features*. Figure 4 illustrates this situation.In this case, the feature LightMng below RoomFacilities appears two times due to the Room feature has been cloned twice. This implies that LightMng, as well as the other features below Room, also evaluate to a set of features instead of simply to true or false. Nevertheless, it should be noticed that these multiple features, oppositely to clonable features, can be evaluated to true or false if they are evaluated in a particular context. For example, the LightMng feature in Figure 4 can be evaluated to true or false when it is evaluated in the context of a particular room. So, our last research challenge is how to deal appropriately with *multiple features*.

Summarising, when dealing with clonable features, we should to address the following research challenges to properly express and analyse cross-tree constraints between features:

1) What a clonable feature means in a cross-tree constraint.
2) Add quantification mechanisms to cross-tree constraints.
3) Add the notion of *context* to cross-tree constraints.
4) Manage properly multiple features.

To the best of our knowledge, there is currently no research work, and consequently no tool, addressing these challenges. To overcome this limitation, we have firstly created a language for expressing arbitrary complex constraints including clonable features as a stepping stone towards the analysis of constraints including clonable (or multiple) features. Next section describes sauch a language.

III. A LANGUAGE FOR EXPRESSING CONSTRAINTS INCLUDING CLONABLE/MULTIPLE FEATURES

This section presents the language we propose for expressing constraints on feature models including clonable features. Figure 5 shows the syntax, in EBNF notation, for such a language.

A *constraint* is a logical expression that evaluates to true or false. A constraint can be simply a literal, i.e true or false (Figure 5 line 00), which evaluates to true and false, respectively. A constraint can also be a simple feature, i.e. a feature that can appear only once as a maximum in a given context. A SimpleFeature evaluates to true if it is selected, otherwise, it evaluates to false.

Clonable features and multiple features are represented in the syntax as MultiValueFtr. A MultiValueFtr evaluates to a positive integer (zero included). This positive integer represents the number of clones of that feature contained in a given context of a configuration model. Since they are numbers, we can use comparison operators, more specifically $<, <=, =, >=, >$ to construct comparison expressions on the number of existing clones. In addition, we can also use these numerical values on basic arithmetic expression, i.e. we can sum, substract, multiply and divide these numerical values. These arithmetic expressions can be used

```
00 <Constraint> ::= "true" | "false" | <SimpleFeature> | <Constraint> <BinaryOp> <Constraint> |
01                  <UnaryOp> <Constraint> | "(" <Constraint> ")" | <ContextExp> | <ComparisonExp>;
02 <BinaryOp>   ::= "and" | "or" | "xor" | "implies";
03 <UnaryOp>    ::= "not";
04 <ContextExp> ::= <SimpleFtr> "[" <Constraint> "]" | "all" <MultiValueFtr> "[" <Constraint> "]" |
05                  "any" <MultiValueFtr> "[" <Constraint> "]";
06 <ComparisonExp> ::= <NumericalExp> <ComparisonOp> <NumericalExp>;
07 <ComparisonOp>  ::= "<" | "<=" | "=" | "=>" | ">" | "!=";
08 <NumericalExp>  ::= <MultiValueFtr> | "SimpleArithmeticExp" | <MultiValueFtr> "[" <Constraint> "]";
```

Figure 5. Syntax of the Hydra Constraint Language

as subexpressions, or operands, in comparison expressions. The comparison expressions evaluate to true or false. Thus, we can use a comparison expression as a subexpression of a more complex logical expression. This solves the first challenge we identified in previous section, which was how to evaluate clonable and multiple features.

Using the language of Figure 5, the context to evaluate a constraint can also be specified. This can be made in several ways. A context can be specified by surrounding a constraint with brackets and given the name of a feature at the beginning of that expression (Figure 5, lines 04-05, line 08 at the end). The feature used as context can be a simple feature or, instead, it can be a clonable/multiple feature. In the first case, the constraint placed into brackets is evaluated using the subtree of the configuration model with root in the simple feature used as context.

In this second case, we can use the operators all and any. If all is used, the constraint enclosed in brackets must be true for all the instances of the MultiValueFtr used as context for the expression being true. Otherwise, it evaluates to false. If any is used, the constraint enclosed in brackets must be true for one instance of the MultiValueFtr used as context at least. If such an instance does not exist, the constraint expression evaluates to false. For instance, any Room[LightMng] would be evaluated to true for the configuration model of Figure 2, whereas all Room[LightMng] would be evaluated to false. This solves the second research challenge identified in the previous section, which was how to deal with quantification mechanisms.

Moreover, none of these operators might be used. In this latter case, the context expression evaluates to the number of instances of the MultiValueFtr feature used as context for which the enclosed constraint is true. For instance, the context expression Room[LightMng] would evaluate to 2 for the configuration model depicted in Figure 2, since LightMng has been selected in two rooms, more specifically, in the Kitchen and in the Bedroom. This, as well as the contents of the previous paragraph, solve the third challenge described in the previous section, which was how to deal with contexts.

We would like to highlight that a feature can be simple in a given context and multiple in another context. For instance, LightMng (see Figure 1) is simple in the context of GeneralFacilities and multiple in the context of SmartHome. This means that LightMng, according to our syntax, can be used as a SimpleFtr inside the Room context; but it must be used as a MultiValueFtr in the SmartHome context. Therefore, Floor[LightMng] would be not a well-formed constraint, whereas Room[LightMng] would be. Therefore, a tool implementing this language should be aware of these details. This would solve the four research challenge identified in the previous section, which was how to deal with multiple features properly. We have taken into account this detail when implementing this language into our feature modelling tool called Hydra.

We have validated this language by applying it to the SmartHome case study. Table 6 shows the cross-tree constraints we have added to the feature model of Figure 1 to avoid creating invalid configurations. We have also applied it to the case studies mentioned in the introduction[6].

C01 specifies that if SmartEnergyMng feature is selected for the whole house, the LightMng feature and the WindowMng must also be selected, since the first one depends on the latter ones. C05 specifies this constraint must be also satisfied at the floor facilities level for all the floors, and C10 does the same at the room level. C01-C04 specify that when a facility is selected for the whole house, it must also be selected for all the floors. C01-C04 specify the same relationship between the floor and the room levels. C11-C13 indicate when a facility is selected, one instances of the corresponding device to be controlled muts be added to the house. Finally, C14 specifies that for the PresenceSimulation working properly, at least a quarter of the house rooms must have automatic control of the lights. It should be noticed that, to the best of our knowledge, this kind of constraint is not supported by any feature modelling tool or language.

Next section describes how we can translate these expression into a Constraint Satisfaction Problem that can be solved using third-party libraries.

IV. Analysis of constraints including clonable/multiple features

Once we have specified a set of cross-tree constraints for a feature model including clonable feature, we might want, and we should, to analyse them. The most basic and required analysis operation is to decide when a given configuration

[6]These case studies can be found in TODO: provide URL

```
C00 GeneralFacilities[SmartEnergyMng implies (HeaterMng and WindowMng)])
C01 GeneralFacilities[LightMng] implies (all FloorFacilities[LightMng])
C02 GeneralFacilities[WindowMng] implies (all FloorFacilities[WindowMng])
C03 GeneralFacilities[HeaterMng] implies (all FloorFacilities[HeaterMng])
C04 GeneralFacilities[SmartEnergy] implies (all Floor[FloorFacilities[SmartEnergy]])
C05 all FloorFacilities[SmartEnergy implies (HeaterMng and WindowMng)])
C06 all Floor[FloorFacilities[LightMng] implies (all Room[LightMng])]
C07 all Floor[FloorFacilities[WindowMng] implies (all Room[WindowMng])]
C08 all Floor[FloorFacilities[HeaterMng] implies (all Room[HeaterMng])]
C09 all Floor[FloorFacilities[SmartEnergy] implies (all Room[SmartEnergy])]
C10 all Room[SmartEnergy implies (HeaterMng and WindowMng)]
C11 all Room[LightMng implies (Light > 0)]
C12 all Room[WindowMng implies (Window > 0)]
C13 all Room[HeaterMng implies (Heater > 0)]
C14 PresenceSimulation implies ((Room[Light] / Room) * 100 >= 25)
```

Figure 6. Cross-tree constraints for the Smart Home feature model

is a valid configuration, i.e. it satisfies all the specified constraints. Nevertheless, other analysis operations, such as the ones surveyed for Benavides et al [2], are of interest. An example of this operation are the analysis of *promising partial configuration* and *dead features*. The *promising partial configuration* analyses if a partial configuration satisfies the constraints and the features for which a decision have not been adopted yet can be assigned a value in way that all the constraints are satisfied. The *dead features* analysis consist on checking if each feature can be included in one valid configuration at least. Otherwise, such a feature might not have been selected, so it would not make sense to have it included in the feature model. Dead features are symptoms of ill-designed feature models. The input for this analysis process

To be able to carry out this kind of analysis, the expressions constructed using the language presented in the previous section are transformed into a Constraint Satisfaction Problem [10]. We have opted for this formalism due to the reported advantages [2], as well as for the good performance provided by the Choco library for CSP solving [8].

A Constraint Satisfaction Problem is defined as a tuple (X, D, C), where X is a finite set of variables, D is a finite set of domains of values (one domain for each existing variable in X), and C is set of constraints defined on V. Thus, the steps to transform our constraint set into a CSP are: (1) to decide how many variables we have to create; (2) what the domains for these variables are; and (3) how to transform the constraints to express them properly as constraint for a CSP.

For the transformation process, all features will be considered as clonable features. Mandatory ones will have a $[1..1]$ multiplicity; whereas optional ones will have $[0..1]$. Feature groups, which have not been used for the Smart Home case study, are transformed into a set of optional features plus a set of cross-tree constraint for ensuring group cardinality, as usual in the literature [6].

Before creating the variables, the *real lower and upper bound* for each feature are calculated. We say *real* because a feature can have more instances than the upper bound of its multiplicity. Similarly, a feature can have less instances than the lower bound of its multiplicity. Figure 7 illustrates this situation. In Figure 7 (a), the feature C has as upper bound 2, which means it can appear two times per instance of the feature B as a maximum. But in the context of the whole feature model, feature B can appear three times as a maximum, so there can be a maximum of six instances of feature C per feature model. This would be the real upper bound of feature C. The *real upper bound* can be defined as the maximum number of instances of a feature in a configuration model as a whole, instead of in a per feature basis.

Similarly, the lower bound for feature C is 1, which means there must be one instance at least per instance of the feature B. Nevertheless, since feature B might have zero instances, it would possible to create a configuration model with zero instances of the feature C. So, zero would be the real lower bound of feature C. The *real lower bound* can be defined as the minimum number of instances of a feature in a configuration model as a whole, instead of in a per feature basis.

The real upper bound of a feature F is calculated by multiplying the upper bounds of each feature in the path from such a feature F to the root of the feature model. It should be noticed as soon as the upper bound of a feature in this path is infinite, the real upper bound of the feature F will be infinite. This would be the case for feature C in Figure 7 (b). Similarly, the real lower bound of a feature F is calculated by multiplying the lower bounds of each feature in the path from such a feature F to the root of the feature model. It should be noticed as soon as the lower bound of a feature in this path is zero, the real lower bound of the feature F will be zero. This would be the case for feature C in Figure 7 (a).

TODO: Redo de figure accordingly to the text

The algorithm for creating the variables for the CSP is as follows:

1) A variable for each feature in the feature model is

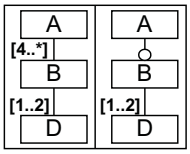

Figure 7. Calculating lower and upper bound of clonable features

created. If the feature can be univocally identified, the name of the variable is the same name as the name of the feature. Otherwise, we preclude the name of the variable with the names of as many ancestor features as required to univocally identified it. For instance, the feature LightMng below GeneralFacilities would be named as GeneralFacilities_LightMng.

2) The real upper bound for each feature is calculated.
3) The domain $[a-b] \in \mathbb{N}$ is associated to each variable. a, b are positive integers (including zero), b can be infinite and $a \leq b$. a is the lower bound of the multiplicity corresponding to the feature the variable represents, and b is the real upper bound for such a feature.
4) For each instance of clonable feature in a given configuration model, a new variable is created. The domain for these variables is $1 \in \mathbb{N}$. The name of the clone is used as name for the variable. Each clone must be given a unique name (this must be ensured by the feature modelling tool).
5) For each instance of a multiple feature in a given configuration model, a new variable is also created. The domain for these variables is $[a-b] \in \mathbb{N}$, where a and b are the *real lower and upper bounds* for that feature, but using as root the nearest clone in the path to the root. The name of the variable is the name of the feature preceded by the name of the first clone found in the path from the multiple feature to the root of a configuration model.
6) Finally, we need to bind the variables that have been already selected or unselected in the configuration model. For each boolean variable v, a constraint eq(v,true) is added to the set of constraints if the corresponding feature has been selected; otherwise, a constraint eq(v,false) is added to such a set. For each integer variable representing a clonable or multiple feature, the number of clones is calculated and that variable initialized to that number of clones.

Once we have created the variables and the domains, the remaining step is to express the constraints in a suitable form for being managed as a CSP. We would need to express these constraints in Choco syntax. Choco is the third-party Java library we have selected for CSP solving due to the promising performance it has shown in several benchmarks [8]. Choco provides logical operators plus comparison

```
GeneralFacilities_SmartEnergyMng implies
    (GeneralFacilities_HeaterMng and
      GeneralFacilities_WindowMng).
```

Figure 8. Constraint C01 rewritten in Choco syntax

```
and(implies(Kitchen_SmartEnergy,
  and(Kitchen_HeaterMng, Kitchen_WindowMng)),
   and(implies(Living_SmartEnergy,
and(Living_HeaterMng, Living_WindowMng))),
  implies(Bed_SmartEnergy,
and(Bed_HeaterMng,Bed_WindowMng)))
```

Figure 9. Constraint C10 expanded rewritten in Choco syntax

operators for specifying constraints. Thus, the problem of translating logical expressions and comparison expressions can be reduced to rewriting the constraints specified in the language presented in the previous section in proper Choco syntax.

Thus, our problem is basically reduced to the translation of context expressions. The solution for this case is to create a new constraint for each clone which serves as context for a expression or subexpression. The new constraint is rewritten for that particular context. For instance, constraint CO1 (see Figure 6) would be rewritten as shown in Figure 8.

If quantifiers are used, the generated constraints are connected by *and* or *or* operators depending on if the all or the any quantifier has been used. For the cases where a context expression evaluates to a positive integer, we simply count how many generated constraints are satisfied.

For instance, in the case of the constraint C10 (see Figure 6) and the configuration model of Figure 2, three constraints would be generated, one per each room instance. These constraints would be rewritten to particularize it for each clone and the they will be connected by *and* operators, as shown in Figure **??**

Once we have defined our CSP, we can solve it using a third-party library as Choco. Choco calculates if the current configuration satisfies the external constraints, and if it is not so, it provides information about what constraints has been violated. Moreover, Choco can be used to perform some kind of useful analysis on partial configurations, calculating if exist one solution at least for the given CSP. This is useful to analyse when a partial configuration is promising, i.e. when it is part of a valid product. It can be also used to perform *dead features analysis*. For each feature, we would calculate if there is one solution at least for the CSP problem with such a feature selected. If this solution is found, the feature is included in at least one valid product. Otherwise, that feature can not be selected and it is considered dead.

Next section comments on the experiments we have carried out for validating our approach and discuss briefly on strengths and weaknesses.

V. Evaluation and Discussion

VI. Related Work

There are currently several tools that supports feature modelling. Table **??** contains a high-level comparison of the most well-known feature modelling tools regarding support for clonable features and usability of the user interface. Most of them supports automatic validation of external constraints defined between features. The common technique used for validating a configuration of a feature model is to transform the feature model and the externally defined constraints

But, as it will be described in this section, there is no tool, to the best of our knowledge, that supports validation of constraints involving clonable features.

Feature Modelling Plugin (FMP) [3] is an Ecore-based [] Eclipse plugin that supports modelling of cardinality-based feature models. Nevertheless, the configuration of clonable features is not supported at all (tool authors acknowledge that configuration facilities are currently unpredictable). FMP supports the specification of external constraints in the form of propositional logic formulas, but the semantics of clonable features in these constraints are simply unspecified. FMP uses JavaBDD [] as a back-end for analyzing and validating constraints in feature models. JavaBDD is an open-source Java library for manipulating Binary Decision Diagrams (BDDs), which are used to analyse the satisfiability of a set of propositional formulas, i.e., of a set of feature relationships and external constraints represented as a set of external formulas. Moreover, the Graphical User Interface (GUI) of FMP is based on the default tree-based representation of Ecore Models, which hinders usability and understability of feature models, overall when users need to deal with large scale models.

Feature Model Analyzer (FaMA) is a framework for the automated analysis [] of feature models. Among the analysis included, we can find ... Nevertheless, FaMA analysis feature models created with other feature modelling tools.

Since the state-of-art feature modelling tools do not support the specification of external constraints involving clonable features, FaMA, to the best of our knowledge, does not incoporate any kind of analysis for validating this kind of constraint.

Moskitt Feature Modeliing (MFM) is a graphical editor for feature models, based on the Mooskkit graphical supports the graphical edition of feature models, including clonable features. Nevertheless, Moskitt only supports the specification of simple binary constraints between features, more specifically, the specification of *requires*, i.e. A implies B, and *excludes* relationships. The semantics of these binary relationships when applied to clonable features is undefined.

TO BE COMPLETED

VII. ConclusionsFuture Work

References

[1] D. S. Batory, "Feature models, grammars, and propositional formulas," in *Proc. of the 9th Int. Conference on Software Product Lines (SPLC)*, ser. LNCS, J. H. Obbink and K. Pohl, Eds., vol. 3714, Rennes (France), September 2005, pp. 7–20.

[2] D. Benavides, S. Segura, and A. R. Cortés, "Automated Analysis of Feature Models 20 Years Later: A Literature Review," *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.

[3] K. Czarnecki, M. Antkiewicz, C. H. P. Kim, S. Lau, and K. Pietroszek, "FMP and FMP2RSM: Eclipse Plug-ins for Modelling Features Using Model Templates," in *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, San Diego (California, USA), October 2005, pp. 200–201.

[4] K. Czarnecki, T. Bednasch, P. Unger, and U. W. Eisenecker, "Generative programming for embedded software: An industrial experience report," in *Proc. of the 1st Int. Conference on Generative Programming and Component Engineering (GPCE)*, ser. LNCS, D. S. Batory, C. Consel, and W. Taha, Eds., vol. 2487, Pittsburgh (Pennsylvania, USA), October 2002, pp. 156–172.

[5] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Formalizing Cardinality-based Feature Models and Their Specialization," *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.

[6] ——, "Staged Configuration through Specialization and Multilevel Configuration of Feature Models," *Software Process: Improvement and Practice*, vol. 10, no. 2, pp. 143–169, January-March 2005.

[7] H. Morganho, C. Gomes, J. P. Pimentão, R. Ribeiro, C. Pohl, B. Grammel, A. Rummler, C. Schwanninger, L. Fiege, and M. Jaeger, "Requirement Specifications for Industrial Case Studies," AMPLE EC Project, Tech. Rep. D5.2, March 2008.

[8] G. Rochart.

[9] A. L. Santos, K. Koskimies, and A. Lopes, "Automated Domain-Specific Modeling Languages for Generating Framework-Based Applications," in *Proc. of the 12th Int. Conference of Software Product Lines (SPLC)*, Limerick (Ireland), September 2008, to appear.

[10] E. Tsang, *Foundations of Constraint Satisfaction*. Academic Press, August 1933.