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 V comments on related work. Section VI 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.

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.

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

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 model would be more useful for this case study than a feature. We will skip this question at this moment and we will come back to it in the discussion section.

Nevertheless, 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). Moreover, we might want to specify more advanced constraints, like if Presence Simulation is going to be included in a automated house, at least a 30% of the rooms in the house must include automatic light management.

In feature models, the most usual way to express cross-tree constraints is using propositional logic formulas, like SmartEnergyMng implies (LightMng and HeaterMng), where the different features are the atoms for these formulas [1].

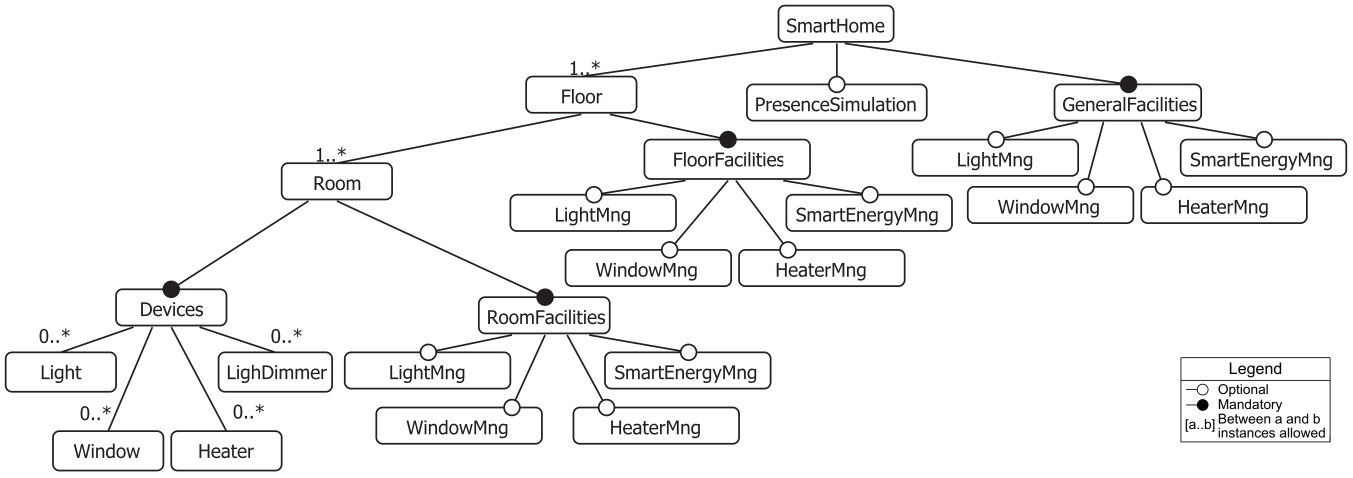

Figure 1. Feature model for a SmartHome SPL

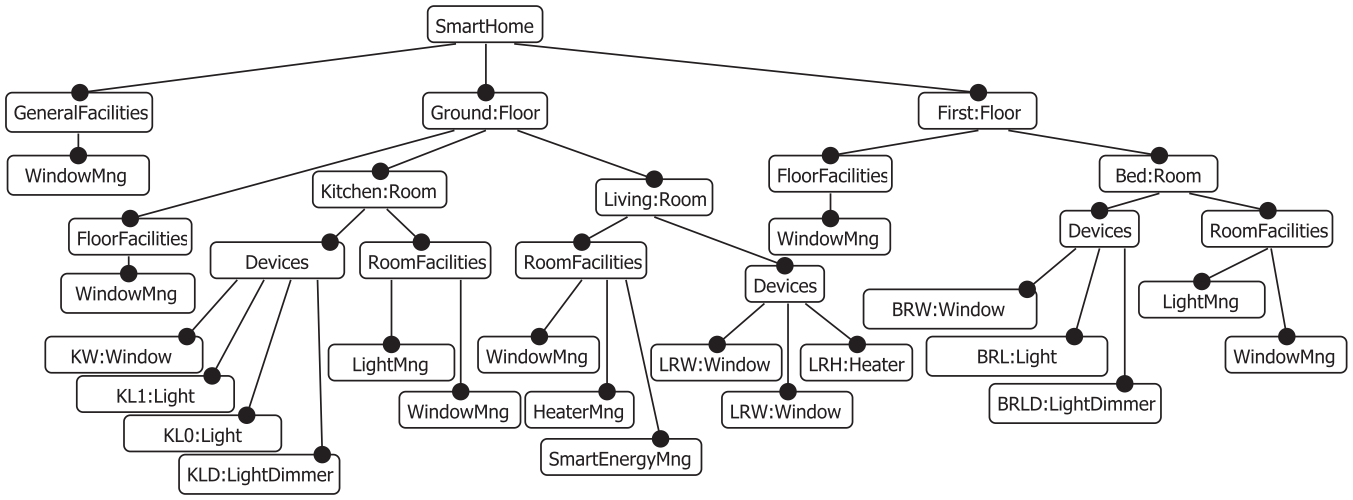

Figure 2. Configuration of a specific automated house

A feature is evaluated to true when it has been selected, and false otherwise. The main problem when dealing with clonable features is that this correspondence is not useful anymore. Clonable features are not selected or unselected, they are simply cloned. So, the meaning of a atom related to clonable feature becomes undefined and, as a result, the formula can not be evaluated.

For instance, let us suppose A and B are clonable features. In this case, what would be the semantics 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, otherwise, does it means that the existence of all potential As implies the existence of all potential Bs? So, the first research challenge we face is to decide what a clonable feature means in a logical formula.

Since clonable features has associated a set of clones, we might want to specify properties that only applies to: (1) at least one element of the set; (2) to all the elements of the set that fulfill a certain constraint; or (3) all the elements. For instance, we might want to specify constraint such as: if HeaterMng is selected per house level, at least one Room must contain a Heater. So, the second research challenge is to add quantification mechanisms to constraints involving clonable features.

Moreover, 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 as facility for a particular Room, such a Room must have at least one clone of Light. If we specify a constraint "LightMng implies at least one Light",

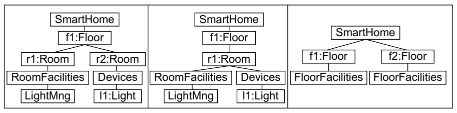

Figure 3. (a) Invalid configuration (b) Valid configuration (c) Multiple features

but we do not limit the scope in which this constraint is evaluated, this constraint might 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 Light to control. This means this constraint must be evaluated for all rooms (notice we need again quantification) and using only the subtree below each Room. Thus, the third research challenge is to specify the context where each constraint must be evaluated.

Finally, when dealing with clonable feature, not only clonable features can appear more than one time in a configuration model. A feature can appear more than one time one of its ancestors is clonable. Figure 3 (a) illustrates this situation. We call to a feature that can appear more than one time a *multiple feature*. For instance, the feature FloorFacilities, which is not clonable itself, might appear several times in a configuration model as a consequence of the Floor feature being cloned. This implies that FloorFacilities also evaluates to a set of features when it is evaluated in the context of the entire configuration model. Nevertheless, it should be noticed that this feature can be evaluated to true or false if it is evaluated in the context of a Floor feature, since it can appear only once in that context. So, a last research challenge is how to deal appropriately with *multiple features*.

Summarising, when dealing with clonable features, we need to address the following research challenges in order to properly express constraints between features:

1) What a clonable feature means inside a constraint expression.
2) Add quantification mechanisms to constraints expressions.
3) Add the notion of contexts to constraints expressions and subexpressions.
4) Design a mechanism for properly dealing with clonable features.

State-of-art tools only offer two operators, implies and excludes, and deal with simple propositional formulas. This is clearly not enough to address the research challenges described in this section. To solve this limitation, we have created an expressive language for expressing arbitrary complex constraints in a user-friendly fashion and we have implemented a reasoner able to decide if a set of external constraints, involving clonable or multiple features, is satisfied given a specific configuration of a feature model. The reasoner is also able to perform some extra task, such as deciding which features must be incorporated to a configuration in order to satisfy the external constraints. We have integrated this language and the reasoner in our feature modelling environment, which we have called *Hydra*.

```
<Constraint> ::= "true" | "false" | <SimpleFeature> | <Constraint> <
                 <UnaryOp> <Constraint> | "(" <Constraint> ")" | <Co
                 <ComparisonExp>;
<BinaryOp>   ::= "and" | "or" | "xor" | "implies";
<UnaryOp>    ::= "not";
<ContextExp> ::= <SimpleFeature> "[" <Constraint> "]" |
                 "all" <MultiValueFeature> "[" <Constraint> "]" |
                 "any" <MultiValueFeature> "[" <Constraint> "]";
<ComparisonExp> ::= <NumericalExp> <ComparisonOp> <NumericalExp>;
<ComparisonOp>  ::= "<" | "<=" | "=" | "=>" | ">" | "!=";
<NumericalExp>  ::= <MultiValueFeature> | "PositiveInteger" |
                    <MultiValueFeature> "[" <Constraint "]";
```

Figure 4. Hydra Constraint Language Syntax

Next section describes the language for expressing external constraints involving clonable features and the reasoner that analyse the satisfiability of these constraints.

III. Expressing and validating constraints involving clonable features

This section presents the language we propose for expressing constraints on feature models involving clonable features and we explain how we can validate these constraints by transforming them into a Constraint Satisfaction Problem (CSP).

A. Expressing external constraints with clonable features

Figure 4 contains the syntax of the language we propose for expressing constraints involving clonable features on feature models. A *constraint* is a logical expression that evaluates to true or false. A constraint can be simply a literal true or false (Figure 4), which evaluates to true and false, respectively. A constraint can also be a simple feature, i.e., a feature that can appear only once at maximum in a certain context. A SimpleFeature evaluates to true if it is selected and, otherwise, to false.

We call to those features that can appear more than once in a given context MultiValueFeatures. These features are clonable features and multiple features, i.e., features which are not clonable but that can appear more than once because they a clonable ancestor, and therefore the subtree where they are placed can be cloned. A MultiValueFeature evaluates to a positive integer, more specifically to the number of clones that currently exist of that feature in a given context of the configuration model. Then, we can use comparison operators, more specifically $<, <=, =, >=, >$ to compare on the number of existing clones. This comparison operators evaluates to true or false. Thus, we can embed this comparison expression into more complex logical expressions. This solves the first challenge we identified in previous section, which was how to evaluate clonable and multiple features.

We can also specify the context that will be use to evaluate a given constraint. This can be made in several different ways. A context is specified by surrounding a constraint with brackets and given the name of a feature at the beginning of the expression (Figure 4, lines 05-07 and line 11). There are several alternatives to evaluate a context expression. If the feature that serves as context is a simple feature, the constraint placed into brackets is evaluated using the subtree of the configuration model with root in the simple feature as context. The result of the ContextExpression is the result of evaluating the Constraint.

Otherwise, the feature that specifies a context is a multivalue feature. In this case, the constraint is evaluated using as context the subtree with root in each existing clone of the multivalue feature. A context expression with a mutilvalue feature as contexts evaluates to the number of subtrees for which the constraint evaluates to true. For instance, in the context expression Room[LightMng], using the configuration model depicted in Figure 2, LightMng would evaluated using the subtrees with root in Bedroom and Kitchen and it will evaluate to 1, since LightMng is selected only in one room. Our feature modelling tool, called *Hydra* checks that each name refers exclusively to only one feature. If different features share the same name in the feature model, they need to be disambiguated using contexts. This solves 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, i.e. the whole feature model, as it was already mentioned in the previous section. This means that LightMng, according to our syntax, is a valid constraint in the Room context, but and invalid expression in the SmartHome context. Thus, LightMng or Floor[LightMng] would be not valid sentences of our language, whereas Room[LightMng] would be. Hydra takes care of this by means checking of what kind each feature is in a given context. Basically, a feature is a multivalue feature if: (1) it is a clonable feature; or (2) in a given context, one of their ancestors is a clonable feature. Then, Hydra checks we are not using multivalue features as terminal symbols and that each multivalue feature is embedded in a ComparisonExpression which returns a boolean value at the end. This solves the four research challenge identified in the previous section, which was how to properly deal with multiple features.

Finally, in context expression with multivalue features, we can also use quantification operators all and any to specify the number of clones of that feature for which the specified constraint must be true. Context expression using quantifiers evaluates to true or false. An expression quantified by any evaluates to true, if the constraint evaluates to true for at least the context provided by one clone of the multivalue feature. Otherwise, it evaluates to false. An expression quantified by all evaluates to true, if the constraint

```
Facilities[SmartEnergy implies (HeaterMng and W
Facilities[LightMng] implies (all Floor[FloorFa
Facilities[WindowMng] implies (all Floor[FloorF
Facilities[HeaterMng] implies (all Floor[FloorF
Facilities[SmartEnergy] implies (all Floor[Floo
all Floor[FloorFacilities[SmartEnergy implies (
all Floor[FloorFacilities[LightMng] implies (al
all Floor[FloorFacilities[WindowMng] implies (a
all Floor[FloorFacilities[HeaterMng] implies (a
all Floor[FloorFacilities[SmartEnergy] implies
all Room[SmartEnergy implies (HeaterMng and Wir
all Room[LightMng implies (Light > 0)]
all Room[WindowMng implies (Window > 0)]
all Room[HeaterMng implies (Heater > 0)]
all Room[LightMng or HeaterMng or WindowMng]
```

Figure 5. Constrains involving clonable features

evaluates to true in each contexts provided by all the clones of the multivalue feature. Otherwise, it evaluates to false. For instance, any Room[LightMng] would evaluate to true using the configuration model of Figure 2, whereas all Room[LightMng] would evaluate to false. This solves the second research challenge identified in the previous section, which was how to deal with quantification mechanisms.

We have validated this language by applying it to the SmartHome case study. Table 5 shows the different external constraints we have added to the feature model of Figure 1 to avoid creating invalid configurations.

Next section describe how we can translate these expression into a Constraint Satisfaction Problem that we can solve with the help of third-party libraries.

B. Validation of external constraints with clonable features

Once we have specified a set of external constraints, we need to design a mechanism for evaluating them given a certain input configuration model, in order to decide if such a configuration model satisfies these constraints and it can, therefore, be considered a valid configuration. Moreover, if a configuration were not valid, we would like to know why it is not valid, and, if it is possible, to (semi)automatically carry out some corrective actions. The input configuration model can be a partial configuration model, i.e. a configuration model where certain features has still neither selected nor unselected.

We can evaluate these constraints and achieve these goals by translating them into a Constraint Satisfaction Problem [10]. A Constraint Satisfaction Problem is defined as a triple (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, we need to decide how many variables we need to create, which domain these variables will have and if we need to adapt our constraints in order to fit in with a constraint satisfaction problem.

A first tentative, is to create a variable by each potential feature, and to assign to each variable a boolean value depending on if the variable has been selected or not. Nevertheless, since clonable features can have an infinite upper bound, there might be an infinite number of variables, and the set of variables X must be finite. But, it should be noticed that although a feature model can have an infinite number of configurations, each configuration is finite, since each configuration must have, by definition, a finite number of clones. Therefore, clonable and multiple features are translated into variables of a CSP based on a (partial) configuration model, instead of a configuration model.

The algorithm for creating the variables for the CSP is as follows. We assume that each clone has a unique name, which serves as feature identifier.

1) Then, we create a variable for each simple feature in the feature model. The domain of each variable is $true, false$. If a feature can be univocally identified, the name of the variable is the same name as the feature. Otherwise, we preclude the name of the variable with as many name of ancestor features were required to univocally identified it. So, a feature referenced as $Facilities[LightMng]$ would be translated into a variable with name Facilities_LightMng
2) For each clonable or multiple feature, we create a variable with domain $a..b$, where a, b are positive integers, and b can be infinite. These boundaries are calculated using the process that will explain below.
3) For each clone in the configuration model, we create a variable for each multiple feature that becomes a simple feature in the context of such a clone. The domain of each variable is $true, false$. The name of the variable is the same name of the feature, preclude by the name of the clone. So, the variable for the FloorFacilities feature belonging to the GroundFloor clone would be name as GroundFloor_FloorFacilities. As before, we preclude the name of the variable with as many name of ancestor features were required to univocally identified it.
4) For each clone, we create a variable for each multiple feature that remains a multiple feature in the context of such a clone. The domain of each variable is $a..b$, where a, b are positive integers, and b can be infinite. These boundaries are calculated using the process that will explain below, but using the clone as a root for the feature model.

If a clonable feature has as cardinality $a..b$, it does not mean that this feature can have between a and b instances. See for instance Figure 6 (a). Feature C has as lower bound 1, but it means it need to appear at least one time by each feature B, and there must be four clones of the feature

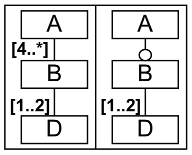

Figure 6. Calculating lower and upper bound of clonable features

B at least. Therefore, the minimum number of clones of the feature C in a whole configuration model is four. So, the global 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. In this path, optional features are considered to have cardinality $0..1$, mandatory features $1..1$ and grouped features has the same cardinality as the feature group. So, the lower bound of feature C, in Figure 6 (b) would be zero. Upper bounds are calculated in the same way, but multiplying the upper bounds of each features. For instance, the upper bound of C in Figure 6 (a) would be infinite; and for Figure 6 (a) would be two.

Applying this process to the feature model of Figure 1 and the configuration model of Figure 2 would be as follows:

1) We create a variable for $SmartHome$ and $Facilities$ with domain $true, false$.
2) We create the variables $Floor$, $Room$, $Devices$, $FloorFacilities$, $RoomFacilities$ with domain $[1..*]$.
3) We create the variables $Window$, $Heater$ and $Light$, $LightMng$, $WindowMng$, $HeaterMng$ and $SmartEnergyMng$ with domain $[0..*]$.
4) We create the variables $Facilities_{L}ightMng$, $Facilities_{W}indowMng$, $Facilities_{L}ightMng$ and $Facilities_{S}martEnergyMng$ with domain $true, false$.
5) We create the variables $GroundFloor_{F}loorFacilities$, $GroundFloor_{F}loorFacilities_{W}indowMng$, $GroundFloor_{F}loorFacilities_{L}ightMng$ and $GroundFloor_{F}loorFacilities_{S}martEnergyMng$ with domain $true, false$.
6) We create the variables $GroundFloor_{R}oom$, $GroundFloor_{R}oomFacilities$, $GroundFloor_{D}evices$ with domain $[1..*]$.
7) We create the variables $GroundFloor_{W}indow$, $GroundFloor_{H}eater$, $GroundFloor_{L}ight$, $GroundFloor_{L}ightMng$, $GroundFloor_{W}indowMng$, $GroundFloor_{H}eaterMng$ and $GroundFloor_{S}martEnergyMng$ with domain $[0..*]$.
8) We create the variables $Kitchen_{R}oomFacilities$, $Kitchen_{R}oomFacilities_{W}indowMng$,

```
\imp{implies(Facilities\_SmartEnergy,and(Facilities\_HeaterMng,Facilities\_WindowMng))}
```

Figure 7. CSP constraint for constraint C01

```
implies(GroundFloor\_FloorFacilities\_SmartEnergy,
        and(GroundFloor\_FloorFacilities\_HeaterMng,
            GroundFloor\_FloorFacilities\_WindowMng))}
```

Figure 8. CSP constraint for constraint C05

$Kitchen_RoomFacilities_LightMng$ and $GroundFloor_RoomFacilities_SmartEnergyMng$ with domain $true, false$.

9) We create the variables $Kitchen_Window$, $Kitchen_Heater$, $Kitchen_Light$ with domain $[0..*]$.
10) We create similar variables to steps 8 and 9, but for the Bedroom clone.

Then, we translate the different constraints into constraints for a CSP. For solving a CSP, we have opted for using Choco [], a Java library for CSP, since it shows a promising performance in several CSP bechmarks []. Choco provides logical operators plus comparison operators for specifying constraints. Thus, the problem of translating logical expressions and comparison expressions is reduced to rewriting the constraints specified in the language presented in the previous section in proper Choco syntax. For instance, the constraint C01 of Figure 5 would be translated into a Choco constraint with the syntax,

Thus, the problem is basically reduced to the translation of context expressions with quantifiers. In this case, the solution for translating a constraint like ¡quantifier¿ ¡MultiValueFeature¿ [Constraint], is to replicate the translation of ¡Constraint¿ as many times as clones the ¡MultiValueFeature¿ has. The translation of ¡Constraint¿ is performed as any other constraint, but assuming that each replica of ¡Constraint¿ is evaluated in the context of an unique clone of such a feature, i.e. the constraint is evaluated using exclusively the subtree below that clone. Therefore, we need to replace the variables in ¡Constraint¿ by the variables that refer to the features below the clone. If the quantifier is any, we join all these replicas by *or* relationships. If the quantifier is all, we join all these replicas by *and* relationships.

Thus, for instance, the constraint C05 in Figure 5, would be translated following the next process: (1) We calculate the set of clones of the feature $Floor$. In this case, $Floor = GroundFloor$. Thus, each feature in the internal constraint refers to a feature below $GroundFloor$; (2) Then, we translate the internal constraint, which generates a Choco constraint such as depicted in **??**.

In the case of the constraint C13 in Figure 5, we would need to create several replicas of the constraint, and to use different sets of variables for each replica, according to the different clones of the feature Room. The translation of such

```
and(implies(Kitchen\_HeaterMng,gt(Kitchen\_Heater,0)),
    implies(Bedroom\_HeaterMng,gt(Bedroom\_Heater,0))))}
```

Figure 9. CSP constraint for constraint C05

a constraint is depicted in Figure **??**

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 defined our CSP, we can solve 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.

For instance, given an invalid partial configuration, Choco can use constraint propagation to calculate what features should be added to the current configuration in order to create a valid configuration. We can also Choco to complete a configuration given a certain criteria.

For instance, let us suppose we have the configuration model of Figure 2, but WindowMng has not been selected neither at the Floor nor at the Room level (and it has been selected at the house level). But, according to constraints C02 and C07, it should have been also selected at the floor and room levels. Using constraint propagation, Choco can calculate that these features are lacking at these levels, and *Hydra*, our feature modelling tool, would add them to the configuration model.

If, given a partial configuration, this can be completed in several ways, we can use Choco to select the configuration, which fulfill some kind of arbitrary criteria, such as having the lower number of features.

IV. Tool Support: Hydra

V. 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

VI. Conclusions, Discussion and Future 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.