A backtrack-free process for deriving product family members

Homero M. Schneider[1]

Abstract. In this paper, we present a new approach for the customisation of product families. It is based on a knowledge framework for representing product families that combines a generic product structure and an extension of the classical constrain network model by the attachment of design functions to the variables. We also present a method for deriving family members from this framework, which consists of a two-stage process. First, a solution to the constraint network is found which is consistent with the set of customer requirements. Second, the solution is used to transform the generic structure into a specific one corresponding to a product family member that meets the customer requirements. One major outcome of the design functions is the establishment of instantiation patterns that guide the problem-solving process. Moreover, if a few modelling conditions are satisfied, it can be proved that finding solutions becomes a backtrack-free process. As a practical example, this approach is used for the implementation of a prototype configurator for a solar powered pumping system.

1 INTRODUCTION

Since the proposal made by Mittal and Frayman [1] to represent product configuration as a CSP problem, many extensions have been put forward to cope with the specificities of configuration problems [2]. Moreover, to improve the efficiency of the product configuration process, it is a practice to use knowledge about the problem domain to guide the search process [3]. Following this rationale, this paper presents an approach to derive members of a product family that exploits the specificities intrinsic to this concept.

It is well known that the design of a product family is a "difficult and challenging task" [4], for it requires the development of multiple products at the same time. However, after the product family is designed, it should not be a surprise that the process of deriving its members can be turned into a routine design task. This claim follows from the fact that during the design process, designers acquire a great amount of knowledge regarding the product family architecture, how the variable aspects depend on each other and their range of variability.

The approach presented in this paper is based on a knowledge framework which combines two general models. A generic product structure (GPS) that represents the product family architecture, and a constraint network model extended with design functions (CN-F) to complement the GPS in the definition of the product family members. The CN-F model is an extension of the classical constraint network (CN) model by the attachment of design functions to its variables. The primary role of these functions is to generate the values for the variables to which they are attached during the customisation process. However, design functions are also used to elicit the dependencies between the variables to form dependency patterns.

In our approach, members of the product family are derived from the knowledge framework as instantiations into two stages. First, a solution to the CN-F model has to be found from the customer requirements. This process is guided by dependency patterns. Then, the solution obtained is used to transform the GPS into a specific physical model that corresponds to a product family member, one that meets the customer requirements.

Although the instantiation patterns can restrict the design space to relatively few variables, they cannot avoid backtracking. Thus, another important contribution of this work is the setting up of modelling conditions such that if the CN-F model satisfies them, the instantiation process becomes backtrack-free. These conditions eliminate the sources of inconsistencies during the execution of the instantiation algorithm proposed for the CN-F model.

In contrast to other approaches that claim to be backtrack-free [5, 6], which typically resort to a pre-processing stage and to computational power, our approach resort to the structuration of the customization process of product families. As a result, it is possible to implement very efficient configurators based on the data flow principle.

As for the remaining of this work, in the next section we review the related literature. In Section 3, we present the SPPS system, which will be used along the paper as our practical example, the solar powered pumping system. In Section 4, we introduce our knowledge framework, by defining the elements of the GPS and CN-F models. In Section 5, we introduce our method for deriving product family members. First, we present our instantiation algorithm. After that, we introduce the conditions for which this algorithm is backtrack-free. Then, we present the method for transforming the GPS into a specific product model. In Section 6, we present the implementation of our prototype configurator. Finally, in Section 7, we make our concluding remarks.

2 RELATED WORK

One early proposal to extend the CSP model was made by Mittal and Falkenhainer [7], who proposed a dynamic constraint satisfaction problem (DCSP) to deal with the fact that the set of variables that are relevant for the solution of a configuration problem may change dynamically during the problem solving. To deal with the structural aspect of configuration problems, Sabin and Freuder [8] proposed a composite CSP. In their approach, the variables are allowed to represent an entire sub problem, such as

[1] Centre for Information Technology Renato Archer, Campinas, Brazil, email: homero.schneider@cti.gov.br

the constituent parts of the final product or the internal structure of components. In [2], Veron et al. proposed to model the configurable product as a tree with internal nodes representing sub-configurable components and leaf nodes corresponding to elementary configurable or standard components. The attributes of the configurable components are represented as variables and each component is associated to a state variable. The configuration process works on two levels. First, the state variables are used to manage the tree structure. Then, the CSP problem is addressed to define the attributes of the active components. The user expresses his choices by adding/retracting unary constraints.

The CSP approaches have been focused mostly on discrete variables and binary constraints. However, in the configuration of engineering products, it is quite common to have continuous variables and constraint on multiple variables. Thus, Gelle et al. [9] introduced local consistency methods to handle discrete and numerical variables and in the same framework to address engineering products represented as a CSP.

With a few exceptions, dependencies have been largely neglected in product configuration approaches. In [10], Xie et al. proposed the Dependent CSP. In this approach, the variables can be related by dependencies or constraints and are divided into independent and dependent by means of the relation of dependency. The independent variables are assigned values from their associated domains, while the values of the dependent variables are assigned values from the values of the independent variables through the relations of dependency. A solution is an assignment to the variables such that all dependencies and constraints are satisfied. The search for solutions is made by a backtracking method of the type "backjunping". The updating of values and the verification of constraints is organized by a directed acyclic graph. This graph is defined based on the dependencies between variables and of constraints in relation to the independent variables. Heuristics are used to establish the order in which variables are considered.

To avoid response delay and dead-ends associated to search-based methods, some recent works resorted to a two-stage process, by precompiling all the solutions using some form of efficient representation. Although these methods still have to solve a hard problem to find all the solutions, this is done offline and only once. Then, the interactive part of the configuration process can be done efficiently. For instance, Hadzic et al. [5] proposed a method to compile all the solutions of the problem using binary decision diagrams. Although they claim that the method has very good practical results, depending on the size of the configuration problem it may run out of space. A different pre-processing method is proposed by Freuder et al. in [6]. Unlike other conventional approaches that add constraints to the problem, thus making them susceptible to space limitation, they remove values from the domain of the variables to make their representation of the problem backtrack-free. The disadvantage of this method is that solutions are lost.

3 THE SOLAR POWERED PUMPING PRODUCT FAMILY

At the core of a solar powered pumping system (SPPS) product family, there is a water pump system and a photovoltaic (PV) array, which provides power to the pump. To improve the pump performance, a pump controller is used to condition the power and to control the pump. A float switch (S_T) is used to turn the pump off when the water tank is full, and another switch (S_W) is used to turn the pump off when the water level at the well is low, thus avoiding that it runs dry. The components of an SPPS are connected by wires to transmit power and control signals. The water is carried from the well to the tank through a piping system. A battery bank may be added to the system if the customer requires the system to have some autonomy, so that water may be pumped at night or during heavily clouded days. A charge controller is used to manage the charging of the battery bank.

Although a typical SPPS is composed of a few components, the product family may have a very large number of variants. For example, the water pump may have many options, each one operating optimally within a narrow window of water head and flux with a specified power, and the PV array can be configured in many ways, based on the choice of the PV model and the arrangement of the components.

Hence, configuring an SPPS to meet the customer requirements and optimizing its performance and cost is far from trivial, demanding a lot of expertise. This precludes most of the potential customers of participating interactively on the decision making along the configuration process, except for providing the application requirements at the beginning of the process.

4 THE PRODUCT FAMILY KNOWLEDGE FRAMEWORK

In the following subsections, we will present our knowledge framework for representing product families. In this approach we assume that the product family has already been developed. However, with this framework we will abstract all the relevant knowledge about the product family for deriving its members.

4.1 The generic product structure

The GPS is a modular architecture composed of component types, which stands for classes of components with the same functionality. In our approach, component types belong to four possible categories: common/generic, optional/generic, common/specific and optional/specific. Figure 1 illustrates schematically the concept of component types and their classification. A component type is specific if the corresponding class has only one component. However, if the corresponding class has two or more components, then the component type is generic.

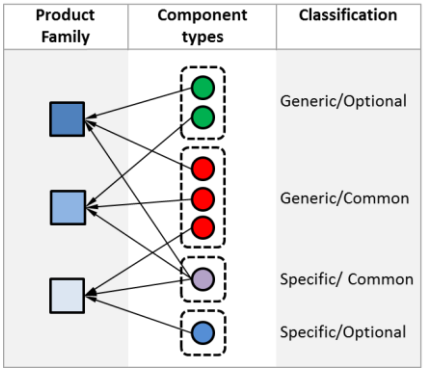

Figure 1. Classification of component types

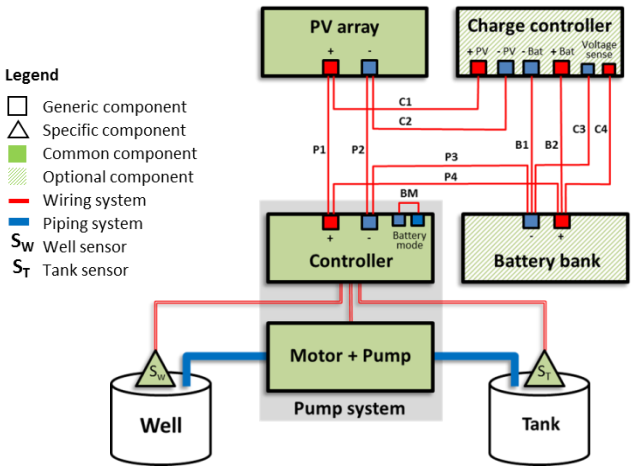

Figure 2..The GPS for the SPPS product family

If all members of the product family have a component in the corresponding class, the component type is common. Otherwise, if at least one member of the product family does not have a corresponding component in the class, it is optional. Note that the component types form a partition on the set of components that is used to derive all the members of the product family.

In Figure 2, it is shown the GPS for the SPPS product family. The PV array, Pump system, Sensors, Wiring and Piping systems are common component types, i.e., they are present in every member of the SPPS product family. However, the Battery bank and Charge controller are optional component types. The Well and Tank sensors are assumed to be specific component types, i.e., they do not vary among applications. All the other components are of the generic type, i.e., they can vary among applications and have two or more variants. It should be noted that, according to our classification, to be a common component type in the product family architecture does not imply that it is fixed. Actually, in our example, most of the product family variability happens on the common part of the GPS. Hence, although the optional components in a product family are one main source of variability, another important source of diversity can be the common part of the product family GPS. This is the case only if it is composed of generic components types.

Formally, we say that a GPS represents the architecture of a given product family if and only if the architecture of each member of that family is isomorphic to a substructure of the GPS and collectively the members of the product family are coherent to the classification of the component types on the GPS.

Hence, given a sample of SPPS, the GPS can be used to decide which of them belong to the product family. On the other hand, the GPS is not enough to determine which configuration of components can lead to a member of the product family, and let alone, which specific configuration will meet the requirements of a given application. To achieve this goal, we combine the GPS with the CN-F model.

4.2 The Constraint Network Extend with Design Functions

The CN-F model used in our approach can be regarded as an extension of the traditional CN model. It is defined by the tuple (V, C, F), where V is a set of variables, C is a set of constraints on subsets of V, and F is a set of design functions (which will be abbreviated as *d*-function), such that, every variable in V has at least one *d*-function attached to it that can generate its values. In what follows, we will define each of these elements and show how they apply to the SPPS product family in complement to the GPS.

Variables – Variations between the members of the product family are identified by variables in V. Consequently, these variables can be mapped on the GPS. Their scope of variation can vary widely, since they may be related from a specific feature to a whole component. For example, the configuration of the PV array is completely specified by three variables: *PV module model*, *PV modules in series* and *PV module strings in parallel*. The pump is associated only to the variable *Pump model*. The range of values that can be assigned to a variable is called its domain. For example, the domain for the variable *Pump model* is composed by the set of pumps {HR-03, HR-03H, HR-04, HR-04H, HR-07, HR-14, HR-20, C-SJ5-8, C-SJ8-7}.

Since all the variability of the product family is related to optional and generic components, only these types of components are associated with variables. These variables will be referred to as output variables because after their values are assigned, a product family member is specified. A special type of output variable is the inclusion variable associated to optional component types (e.g., *Battery inclusion*). These are binary variables that define if the component is included or not in the derived product.

However, variations can also be related to the application environment. For the SPPS example, the amount of *Daily water* needed, the *Well yield*, the *Tank capacity*, the *System autonomy*, etc., are variables that express the customer requirements and are referred to as input variables. Input and output variables are not necessarily disjoint subsets of V. Besides these two classes, the set V may contain auxiliary variables, which are neither input nor output variables. For example, the variable *Total dynamic head* is defined in terms of input variables, and although it is an essential variable for the choice of the pump system, it is not used to specify directly any of the components in the GPS. Therefore, it is classified as an auxiliary variable. In the SPPS example, we have identified 32 mixed discrete and continuous variables. In Figure 3, they appear as nodes of the constraint network, numbered from 1 to

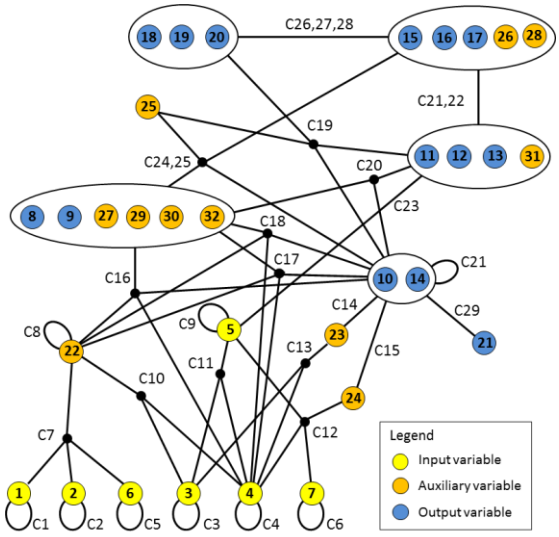

Figure 3. Constraint network for the SPPS product family

32. Some of these variables have been named explicitly within the text. As it will be discussed below, for convenience, variables can be grouped to form a composite variable. The encircled nodes in Figure 3 represent composite variables.

Constraints – Constraints define how subsets of variables in V are related to each other, thus restricting the possible combinations of values that can be assigned to them simultaneously. For example, the following sample of constraints describes how the auxiliary variable *Total dynamic head* is related to some variables in V:

C7: *Total dynamic head* (22) is equal to the sum of the *Water level* (1), *Water drawdown* (2), *Tank elevation* (6) and the friction loss of the piping system.
C8: *Total dynamic head* (22) must be less or equal than the head of the pump system (defined by the combination of the pump and its controller).
C18: If there is a *Battery inclusion* (10), the *Daily water* (4) requirement must be equal or less than 24 hours of pumping with the maximum available *Pump output flux* (32) at the required *Total dynamic head* (22).

Note that while the constraint C8 is defined over one variable, the other two relate four variables. Actually, in our approach, constraints can involve any subset of V. To satisfy a constraint, the values assigned to the variables in the expression defining it must render the expression true. However, if a constraint involves an inclusion variable and the corresponding optional component will not be included in the custom product, it can be disregarded.

Figure 3 depicts the complete constraint network for the SPPS product family. Note that, when nodes are the composition of variables, they may involve more than one constraint, each one relating a different subset of those variables.

Design Functions – The *d*-functions have been introduced as an extension to the CN model to capture the necessary knowledge to generate the values for the variables in V. Generically, *d*-functions will be represented by $f_x(y, z, \ldots, w)$, where x is the depended variable to which the *d*-function is attached and $y, z, \ldots, w$ are the independent variables from which the value for x is generated. As an example, Figure 4 shows the specification of *d*-function F4, which generates the values for *Total dynamic head* as a function of *Water drawdown*, *Water level* and *Tank elevation*.

As we shall see in more details below, an important consequence of *d*-functions is the dependency relation between variables that they establish. However, if the value generated by a *d*-function is to be consistent with the values of the variables it depends on, it must incorporate all the constraints involving these variables. We say that a *d*-function incorporates a constraint if and only if every combination of the values of the independent variables (for which the *d*-function is defined) and the value generated from them, satisfy that constraint. For example, from lines 1, 2, 3 and 4, it can be verified that constraints C7 and C8 are incorporated by F4.

```
F4 (Water drawdown, Water level, Tank elevation)
Begin
1. Geometrical head = Water level +Water drawdown +Tank elevation;
2. Friction Loss = 0.05 × Geometrical head;                          <=> C7
3. Total dynamic head = Geometrical head + Friction Loss;
4. If Total dynamic head ≤ maximum head of the available pump systems => C8
5. Then return Total dynamic head
6. Else notify "The configuration process cannot proceed because there is no
7.      available pump system that can overcome the total dynamic head.";
8.      Abort configuration;
End
```

Figure 4. The *d*-function F4 attached to *Total dynamic head*

In general, not all the variables related (by constraints) to the variable which a *d*-function is attached to will be involved in the dependency. For example, the variables *Daily water*, *Battery inclusion* and *Pump output flux* are related to *Total dynamic head* by the constraint C18 but are not required for the generation of its values. Consequently, C18 is not incorporated by F4. If a *d*-function does not incorporate a constraint involving the variable to which it is attached, we say that the constraint is free regarding that *d*-function. However, a free constraint may be incorporated by another *d*-function attached to the same variable or to a related variable.

Input variables are attached with special *d*-functions that request the user to assign a value chosen from a delimited range of values, which may be generated dynamically as a function of values assigned to other variables. Hence, except possibly for the input variables, all variables in V will necessarily depend on some other variable due to the *d*-function attached to them, forming a network of dependencies on V, as discussed in more detail below.

The *d*-function F4 specified in Figure 4 is relatively simple. The CN-F model for the SPPS also contains much more complex ones. For example, to define the values of the variables that specify the component type PV array (related above), the *d*-function F16 finds the best module arrangement to cope with the power requirements of the SPPS without violating the voltage and current restrictions imposed by the pump or battery controller. As another example, the *d*-function F12 selects the pump system from a performance table which correlates the total dynamic head, the output flux and the input power for the optimal performance of the pump systems.

If a set of variables is strongly coupled, i.e., the value of any one variable cannot be assigned independently of the others, as in the two cases just discussed, they are be grouped together to form a composite variable and the same *d*-function will generate the values for all of them. Otherwise, attaching a single *d*-function to each of those variables would form dependency loops between them, a condition that is undesirable in our approach.

Since only values generated by the *d*-functions are taken into account in the configuration process, in our approach the domain of a variable in V can be defined as the set of all values that can be generated by the *d*-functions attached to it. An important consequence of this definition is that the domains need not to be defined explicitly. Moreover, they can be either discrete or continuous without distinction.

Before introducing the instantiation process for the CN-F model, we note that the dependency between variables in V induces a dependency between *d*-functions in F. For example, the *d*-function F4 attached to *Total dynamic head* depends on the *d*-functions that generate the values to variables *Water drawdown* and *Water level*, *Tank elevation*.

5 DERIVING PRODUCT FAMILY MEMBERS

Members of the product family are derived from the knowledge framework. This process is divided into two stages. First, a solution to the CN-F model is found from the values of the input variables.

Second, this solution is used to transform the GPS into a specific model representing the desired product family member.

5.1 Finding solutions to the CN-F model

An assignment of values to all the variables in V such that no constraint in C is violated is said to be a solution to the CN-F model. The set of all solutions will be denoted by S. As we will argue below, solutions in S correspond to members of the product family.

The instantiation process begins with the assignment of values to the input variables and proceeds towards the output variables, through the auxiliary variables. This process is guided by the dependencies established over V by the d-functions. In Figure 5, we present an instantiation algorithm to carry out this process. In that algorithm, a d-function is enabled if all the variables it depends on have been assigned their values. The set G represents the variables for which the values have already been generated and $L(f)$ represents the set of free variables in relation to f. For this algorithm to work properly, it is necessary to rule out loops between d-functions. Thus, we assume that $F = \{f_1, f_2, \dots, f_k\}$ can be order by the dependency relation induce over F, that is to say, for $i = 1,2, \dots, k$, the element f_i is an input d-function or all d-functions it depends on precedes it in that order.

Every time a d-function $f_x(y, z, \dots, w)$ from F is executed (line 2 of the instantiation algorithm), a value is assigned to variable x from the values of the variables $y, z, \dots, w$. If we represent this dependency by a directed graph, with arrows from the independent variables toward the dependent one, the execution of the instantiation algorithm can be represented by a dependency graph as the one shown in Figure 6. The nodes represent variables (single or composite), the same ones shown on the constraint network in Figure 3. Near to each node, it is indicated the d-function that was used to set its dependency (the incoming arrows). The dependency graph can be organized into dependency levels. At level 0 are the input variables whose values have been assigned by the customer and that do not depend on other variables. In general, a variable is localized at level n if it depends on at least one variable at level $n - 1$. Note that the input variables *Daily water* and *System autonomy*, represented by nodes 4 and 5, appear at levels 2 and 3, respectively. Although it is the customer who assigns their values,

The instantiation algorithm:
Begin
1. Following the order in F, remove the first f which is enabled
2. If f successfully assigns a value to x
3. Then add x to G, remove from F all alternative functions attached to x
4. and make $L = L \cup L(f)$
5. Else If there is another generative function attached to x
6. Then go to step 1
7. Else go to End and return "Failure" ⇒ inconsistency of type I
8. For every $R_X \in L$
9. If $X \subseteq G$
10. Then If R_X is satisfied
11. Then remove it from L
12. Else go to End and return "Failure" ⇒ inconsistency of type II
13. If $F \neq \emptyset$
14. Then go to step 1
15. Else go to End and return "Success"

End

Figure 5. The instantiation algorithm to find solutions to the CN-F model

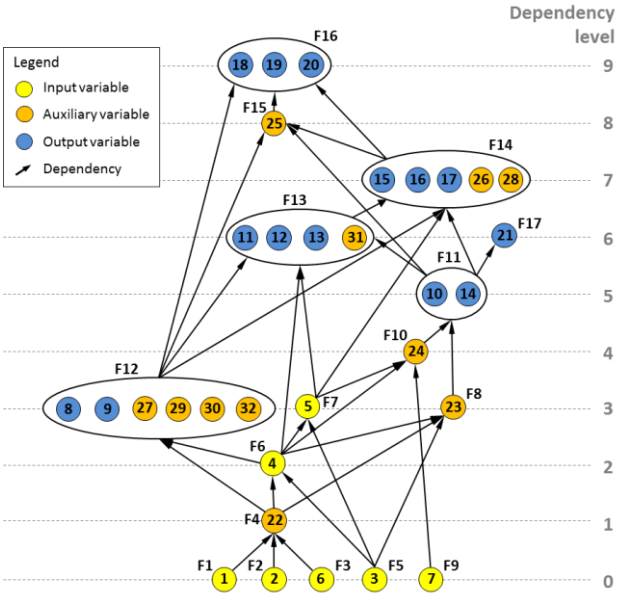

Figure 6. The instantiation graph for the SPPS product family

they also depend on other variables for checking the consistency of the values assigned by the customer.

Now, every instantiation graph can be associated to a subset of F, composed of exactly those d-functions used to generate it. Since the same set of d-functions can be elicited for a variety of inputs, we will call this set an instantiation pattern, represented by P. More specifically, every subset $P \subseteq F$ satisfying the ordering condition and such that, for every $x \in V$, there is only one $f_x \in P$ is an instantiation pattern. If a variable in V is attached with more than one d-function, the CN-F model will be associated to more than one instantiation pattern. However, in general, one should not expect many instantiation patterns. In the modelling of the SPPS example, there is only one instantiation pattern composed of 17 d-functions, number from F1 to F17 in Figure 6.

As indicated in Figure 5, there are only two points during the execution of the instantiation algorithm where it can terminate without finding a solution. Each one is associated to a different type of inconsistency. Type I arises when the d-functions attached to a variable cannot generate its value. The inconsistency of type II, arises if there is a free constraint in $L(f)$ that is violated by the values assigned to the variables in G. If the values assigned to the input variables are not part of a solution in S, then there is some inconsistency embedded in the input and the algorithm will fail.

As it is well known, local consistency in a CN model does not guarantee global consistency [11]. Therefore, although the values generated by the d-functions are locally consistent, the instantiation pattern does not guarantee that an input without an embedded inconsistency will lead to a solution. Thus, in what follows we will introduce two consistency conditions to the CN-F model such that our instantiation algorithm will always be able to find a solution.

Consistency condition 1 – For every $x \in V$, there is at least one $f_x \in F$ which is defined for every instantiation of the variables it depends on.

Consistency condition 2 – Let $P \subseteq F$ be an instantiation pattern. Every constraint in R is incorporated by some d-function belonging to P.

It can be proved that, if the CN-F model satisfies the Consistency conditions 1, no inconsistency of type I will arise during the execution of the instantiation algorithm, and if all its instantiation patterns satisfy the Consistency condition 2, no inconsistencies of type II will arise. However, if the CN-F model satisfies the two conditions, lines 4-12 of the algorithm in Figure 5 can be eliminated, since the inconsistency testing is no longer required. Therefore, the resulting instantiation algorithm becomes extremely simple.

The CN-F model for the SPPS satisfies the second condition state above; however, it fails the first one. The problem is with the *d*-function attached to *Total dynamic head* shown in Figure 4. According to its specification, only after all three inputs variables it depends on have being assigned their values is that the *Total dynamic head* is calculated and the result is compared to the head of the available pumps. If the condition on line 4 is not satisfies, there is no solution to the application and the configuration has to be aborted. To satisfy the Consistency condition 1, an alternative approach is to restrict the range of values for the input variable *Water level* dynamically, so that the resulting total dynamic head of the application is always within the range of the available pump systems. Nevertheless, this restriction is equivalent to the abort condition in a disguised form. On the other hand, because the decision to abort is taken at the very start of the configuration process, and we can give explanations for why the configuration process cannot proceed, this modelling approach was preferred. However, to cope with this abort condition, it was necessary to add a control mechanism in the implementation of the instantiation algorithm, not present in its description in Figure 5. Note that the risk of having to abort the configuration is reduced as the maximum head of the available pumps is increased.

5.2 Transforming the GPS into physical models

Once the solution to the CN-F model has been found, all the output variables on the GPS have their values assigned, and its transformation into a specific physical model can start. This process is carried out in two steps. First, it is necessary to remove the optional components types from the GPS that are not required in view of the customer requirements. For example, if the customer does not require any system autonomy, there is no need for batteries in the SPPS. To determine if an optional component type have to be removed we refer to the value of the associated inclusion variables. In our example, if the value is 0, the component is removed. Otherwise, if it is 1 the component is kept in the structure. After the GPS has been stripped of the unnecessary components, the second step of the transformation process is carried out with the substitution of the generic components by specific ones from their correspondent class of components. The definition of which component will be selected is made based on the values of the output variables on the generic component type. For example, besides the inclusion variable, the Charge controller is associated to three other variables. One of these variables specifies the model of the charger, and the other two the configuration of two switches to set the output voltage of the charger. After all the generic component types have been substituted by specific ones, a physical model of the custom SPPS will emerge from the GSP.

Based on the transformation process described above, every solution in S leads to a specific physical model. Obviously, the resulting physical model is isomorphic to the GPS of the product family and is coherent to the component types by construction. Now, if every relevant design constraint has been elicited and introduced in the CN-F model, we can conclude that every solution in S corresponds to a member of the product family.

6 IMPLEMENTATION OF THE CONFIGURATOR

The SPPS configurator has been conceived as a tool to support the sales force of a company that provides water pumping solutions to the rural area. The configurator requires the sales force to have only enough technical knowledge about SPPS to make some assessments at the customer site to input the customer requirements. This process is interactive with the configurator requesting specific information. To avoid inconsistencies embedded in the input, the configurator makes a few checks, suggesting appropriate corrections if necessary. But in case no solution can be provided to the customer, the configurator notifies the impossibility as early as possible.

In Figure 7, it is shown the implementation of SPPS configurator using LabVIEW. At the centre, it can be seen the *d*-functions (numbered F1 to F17), each one representing a subVI (a kind of routine in LabVIEW), with the variables to which they are attached at the right of the diagram. The variables to which the *d*-functions depend on are indicated by the lines coming from below. Thus, this diagram arrangement clearly reveals the dependency between the *d*-functions. At the left of the diagram, it can be seen the control structure which operates in conjunction with the loop structure (the outer structure encompassing the whole program). Initially, only the first four *d*-functions will be executed. If the abort condition in the *d*-function F4 (specified in Figure 4) is true there is no solution for the configuration problem and the program ends. Otherwise, the abort variable is set to false and the other *d*-functions are executed. As the *d*-functions are executed, the values for the correspondent variables are generated, and they are set to inactive. The *d*-functions attached to variables (other than the inclusion variable) on optional component types, which will not be included in the custom product, can be set to inactive without generating values. When no abortion happens and all the functions are inactive (which is equivalent to $F = \emptyset$ in the control algorithm in Figure 5), a solution has been found and the program ends. This happens in exactly three iterations of this configurator program.

It is interesting to note that, if the CN-F model satisfies the two consistency conditions, the configurator can be implemented a data flow program by the concatenation of *d*-functions. Moreover, if it were not for the abort condition, the iteration structure in Figure 7 could have been dismissed.

7 CONCLUSIONS

In this paper, we have proposed a new approach to the customisation of product families. It is based on a knowledge framework which combines a GPS and a CN-F model to represent product families. Members of the product family are derived from this knowledge framework by a two-stage process. First, a solution to the CN-F model is found from the customer requirements through an instantiation process. Then, in the second stage, the solution is used to transform the product family GPS into a specific model which represents the desired product family member.

A number of contributions to the area of product configuration are introduced by this approach. It is provided a formal definition for the product family GPS and an extension to the classical CN model by attaching *d*-functions to the variables to generate their values. Since the domains of the variables are defined through the *d*-functions, their values need not to be predefined explicitly. As a consequence, we can deal with mixed discrete and continuous variables.

Moreover, the *d*-functions provide a method to establish the dependency between variables as part of the modelling of the customisation process. Dependency patterns can reduce the design space for finding solution considerably. However, despite their local consistent, they do not avoid backtracking. To achieve this goal we have set up a few conditions for the CN-F model, such that, if satisfied, deriving product family members becomes a backtrack-free process. The remarkable aspect about this

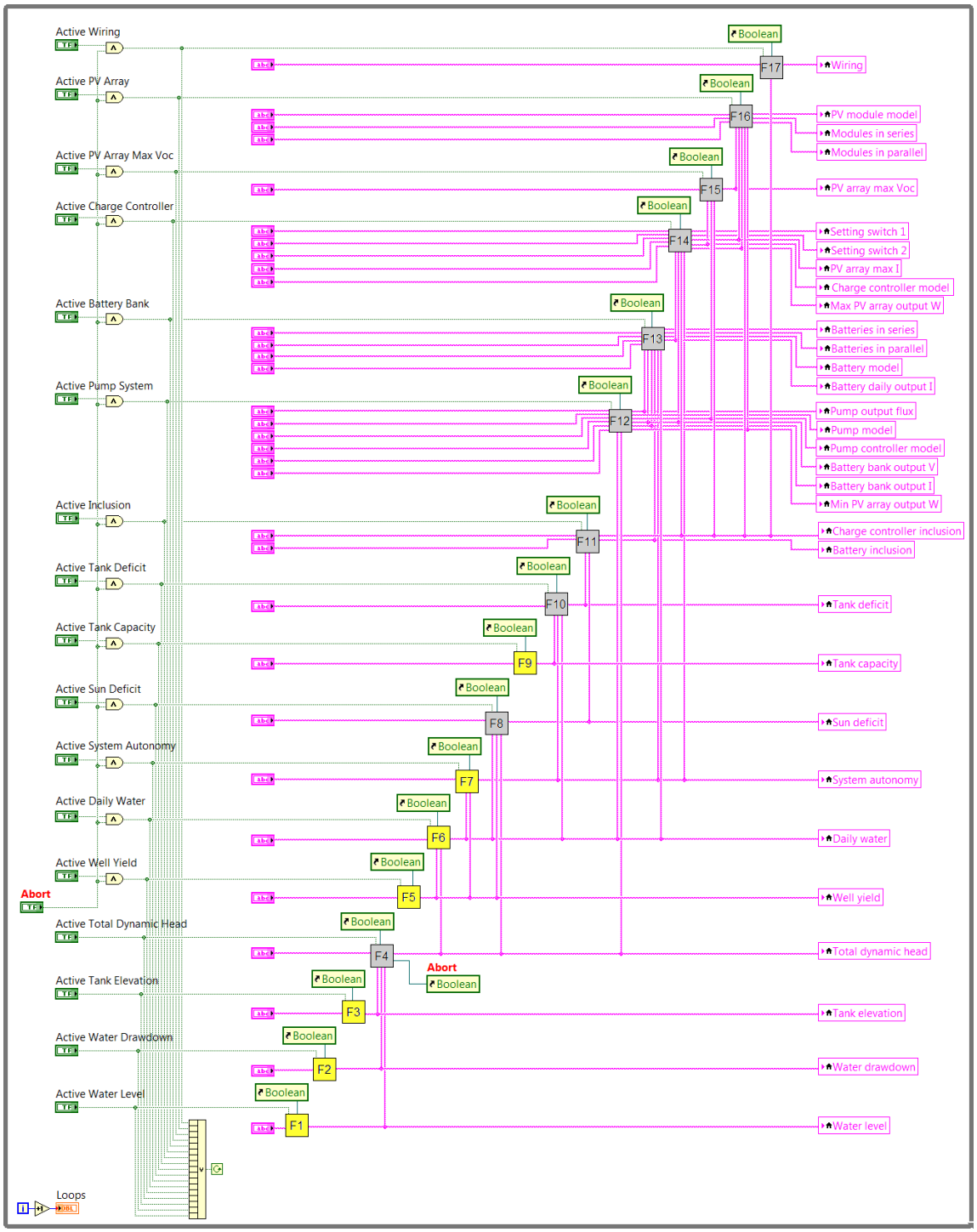

Figure 7. A view of the SPPS configurator program implemented in LabVIEW

achievement is that it does not depend on pre-processing, but can be obtained by the systematization of the knowledge about product families.

It is also interesting to note that through the *d*-functions it may be possible to design components during the customization process, thus providing great flexibility to the customization process. However, this is a capability which requires further investigation, because making changes to components without the proper delimitation of the design space can compromise the manufacturability or performance of the product being derived.

Our approach is suited for the configuration of complex product families for which the customers do not have the necessary expertise to participate directly during all the configuration process. It can deal with configuration problems for which the constraints between the variables are highly complex, since they are incorporated by the *d*-functions and dealt with in the form of procedures. The complexity of the configurator is not particularly affected by the number of variables, since this amounts to adding new *d*-functions. In case some of the variables are attached with more than one *d*-function, this will generate multiple instantiation patterns. However, the proposed instantiation algorithm is enough to deal with this condition, since at every moment only one instantiation pattern is being followed. As for the verification of the compliance to the consistency conditions, this is largely an analysis of the *d*-function individually. (The same is true for maintenance, because *d*-functions are high modular.) Now, if the CN-F model satisfies our assumption on the ordering of the set of *d*-function and the two consistency conditions, the configurators can be implemented in the form of dataflow programs by the concatenation of the *d*-functions.

Despite the advantages related above, to exploit all the potential of our approach in practical applications, there are a number of issues that must be further developed. For example, concerning the integration of our approach into a mass customisation system, it will be necessary to have a more elaborate representation of the GPS to support the generation of customer quotations and production orders [13]. However, at least for a mass customization systems based on 3D printing, we have shown that our approach can be integrated with CAD tools, and that the generation of 3D models for the custom products can be made automatically, in a seamless way [14].

ACKNOWLEDGEMENTS

The author wish to gratefully acknowledge the financial support of FINEP for the realization of this work.

REFERENCES

[1] S. Mittal and F. Frayman, "Towards a Generic Model of Configuration Tasks," in *Proceedings of the 11th International Joint Conference of Artificial Intelligence*, San Francisco: Morgan Kaufman, 1989, pp.1395–1401.

[2] M. Veron, H. Fargier and M. Aldanondo, "From CSP to Configuration Problems," in *AAAI-99 Workshop on Configuration*, Orlando, Florida, July 18–19, 1999.

[3] B. Wielinga and G. Schreiber, "Configuration design problem solving," *IEEE Expert*, vol. 12, no. 2, pp. 49–56, 1997.

[4] T. W. Simpson, B. Aaron, L. A. Slingerland, S. Brennan, D. Logan and K. Reichard, "From user requirements to commonality specifications: an integrated approach to product family design," *Research in Engineering Design*, vol. 23, no. 2, pp. 141–153, 2012.

[5] T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, H. Hulgaard and J. Moller, "Fast backtrack-free product configuration using a precompiled solution space representation," in *International Conference on Economic, Technical and Organizational aspects of Product Configuration Systems*, Technical University of Denmark, Lyngby, Denmark, June 28–29, 2004.

[6] E. C. Freuder, T. Carchrae and J. C. Beck, "Satisfaction Guaranteed," in *Workshop on Configuration, Eighteenth International Joint Conference on Artificial Intelligence*, 2003.

[7] Mittal, S. and Falkenhainer, B., "Dynamic Constraint Satisfaction Problems," in *Proceedings of the 8th National Conference on Artificial Intelligence*, 1990, pp. 25-32.

[8] D. Sabin and F. Freuder, "Configuration as Composite Constraint Satisfaction," in *Technical Report FS-96-03, Workshop on Configuration*, Menlo Park: AAAI Press, 1996, pp. 28–36.

[9] E. Gelle, B. V. Faltings, D. E. Clement, and I. F. C. Smith, "Constraint Satisfaction Methods for Applications in Engineering," *Engineering with Computers*, vol. 16, no. 2, pp. 81–85, 2000.

[10] H. Xie, P. Henderson, J. Neelankavil and J. Li, "A Systematic search strategy for product Configuration," in *17th International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems Manufacturing* (IEA), Ottawa, Ontario, January 1, 2004.

[11] R. Dechter, "Constraint Networks," in *Encyclopedia of Artificial Intelligence*, S. C. Shapiro, Ed. New York, Wiley, pp. 276–285, 1992.

[12] C. Forza and F. Salvador, "Managing for variety in the order acquisition and fulfillment process: The contribution of product configuration systems," *International Journal of Production Economics*, vol. 76, pp. 87–98, 2002.

[13] A. Haug, L. Hvam and N. H. Mortensen, "A layout technique for class diagrams to be used in product configuration projects," *Computers in Industry*, vol. 61, pp. 409–418, 2010.

[14] H. M. Schneider, D. T. Kemmoku, P. Y. Moritomi, J. V. L. da Silva, Y. Iano, "Matching the Capabilities of Additive Technologies with a Flexible and Backtrack-free Product Family Customisation Process," in *Proceedings of the Fraunhofer Direct Digital Manufacturing Conference 2014*, Berlin, Germany, March 12-13, 2014.