Abstract: Feature models are used to represent the variability and commonality of software product lines, and permit the configuration of specific applications. However, universally accepted definitions of feature and feature diagrams are missing. This paper proposes the use of hypergraphs to integrate the different versions of these concepts in an extensible characterization. The definition, validation and selection of feature configurations are based on hypergraph properties and existing algorithms. Once the formalism is stated, the definition of a feature meta-model is straightforward and a set of modeling tools, compatible with the different flavors of feature diagrams, can be readily built. Finally, configuration and transformation of feature diagrams into UML models are defined as algebraic and QVT transformations.

Keywords: Feature model, hypergraph, feature meta-model, CASE tool

1 Introduction

Software product lines constitute a successful reuse paradigm in industrial environments in spite of their complexity [3]. Feature models represent the variability and commonality of software product lines and permit the configuration of each specific application to be selected. However, a universally accepted definition of feature and feature diagram (FD) is missing and many of the variants more frequently used cannot solve some problems. The original proposal, included in the Feature Oriented Domain Analysis (FODA) method [12], defines features as the nodes of a tree, related by various types of edges. The tree root or concept is decomposed by AND, X-OR and OPTIONAL relationships (see the example of Figure 1a), but FODA does not cover disjunction (OR).

Several extensions have been proposed: incorporating the OR decomposition (Figure 1b) [11], changing the visual syntax, or using directed acyclic graphs (DAG) instead of simple trees. However, in spite of using graphs, the situation of Figure 1c is not possible when the multiplicity is a property of the feature. Riebisch et al. [18], for example, proposes moving the multiplicity constraint to arcs instead of nodes. Constraints between features (a feature requires another feature or two features are mutually exclusive) can be added in textual or graphical formats. Schobbens et al.
[19] have evaluated the diverse variant of FDs, clarifying the differences and establishing a generic semantics. The study classifies the existing proposals using several characteristics: the FD is a tree or a DAG, the constraints are textually or graphically shown, and the way the decomposition relationships (AND, X-OR, OR, multiplicity) are expressed. They propose a new non-redundant variant FD or VFD.

Considering these antecedents, we propose the use of directed hypergraphs as a formal structure to define an FD instead of trees or simple graphs. The main reason is to substitute the several types of decomposition (Mandatory/Optional and Alternative/Or groups) and features (Solitary and Grouped Features) with only two elements: features (nodes) and generic decomposition (labeled hyperarcs). An additional advantage is that requires or mutex constraints can be reformulated using hyperarcs as well, contrary to the current proposals. Most authors (see [7] for example) deal with the structural constraints implicit in the features tree (or graph) independently from the additional mutex/requires constraints. The configuration, therefore, has to solve the problems in two stages. Using hypergraphs, the product configuration problem can be reduced to detecting connections and/or hyperconnections in the hypergraph.

The rest of the report is as follows: the next Section introduces hypergraphs and formally defines the structure underlying an FD. Section 3 analyzes the configuration problem and sketches the hypergraph algorithms used for deriving the configuration, starting from a set of features selected by the user. Section 4 uses the formal definition to build an extensible meta-model. In Section 5 a previously defined transformation into UML models is adapted to the proposed meta-model, generalizing tree structures to acyclic graphs. Finally, Section 6 presents related work and Section 7 concludes the paper and considers future work.

2 Hypergraphs and Feature Diagrams

A hypergraph is a generalization of a graph wherein edges can connect more than two vertices and are called hyperedges. Directed hypergraphs extend directed graphs, and
have been used as a modeling and algorithmic tool in many areas: formal languages, relational databases, manufacturing systems, public transportation systems, etc [9]. A technical, as well as historical, introduction to directed hypergraphs has been given by Gallo et al. [9]. The main reason for introducing this type of graphs is to represent Many-to-One relations, for which simple DAG or trees are not well equipped.

A directed hypergraph or simply hypergraph is a pair $H = (V, E)$, where

- $V = \{v_1, v_2, \ldots, v_n\}$ is the set of nodes
- $E = \{e_1, e_2, \ldots, e_m\}$, with $e_i \subseteq V$ for $i=1, \ldots, m$, is the set of hyperarcs

Where a hyperarc is an ordered pair, $e = (t(e), h(e))$, with $t(e) \cap h(e) = \emptyset$. $t(e)$ is the tail of $e$, while $h(e)$ is its head. A Forward hyperarc, or simply F-arc, is a hyperarc $e = (t(e), h(e))$ with $|t(e)|=1$. An F-graph (or F-hypergraph) is a hypergraph whose hyperarcs are F-arcs, that is, all the hyperarcs have only a node as their tail.

2.1 Feature Diagrams as Hypergraphs

A Feature Diagram can be modeled as a directed hypergraph, where the features are the set of the nodes and there is a hyperarc for each decomposition relationship between features. Each hyperarc is assigned a label which corresponds to the multiplicity of the decomposition. The obtained hypergraph is an acyclic labeled F-graph. Each feature decomposition is mapped to an F-arc in the following way:

- mandatory features to hyperarcs where $|h(e)|=1$, and label 1..1
- optional features to hyperarcs where $|h(e)|=1$ and label 0..1
- pure alternative (X-OR) features to hyperarcs where $|h(e)|=q$, with $q>1$ and label 1..1
- OR features to hyperarcs where $|h(e)|=q$, with $q>1$ and label 1..q

The first two situations are the original mandatory and optional relationships. The third is a pure alternative situation and the last one is the generic OR. Therefore, all the possible decomposition variants [19] are covered. To facilitate the validation and later configuration of the FD, constraint relationships are considered formally as additional hyperarcs. The semantics of requires is that the $A$ requires $B$ constraint establishes a compulsory relationship between features $A$ and $B$, i.e. a hyperarc between $A$ and $B$ with label 1..1. In fact, we could generalize to $A$ requires $S$, $S$ being a set of features. The multiplicity minimum and maximum of the hyperarc would be equal, in both cases, to the cardinality of $S$. We would like to point out that introducing this kind of arc (hyperarc) cycles may arise in the hypergraph. This is an undesired and nonsensical situation; therefore, once all the requires hyperarcs have been defined, the acyclicity of the hypergraph could be tested with the F-Acyclic procedure described in [8].

The semantics of mutex is that we cannot select simultaneously more than one of the two or more features involved. Then, independently of the implicit constraints imposed by the graph structure, a new relationship is imposed. This can be reinterpreted as a hyperarc from a common node (the root) to the involved nodes with 0..1 multiplicity: at most one of the features involved can be selected.
Fig. 2 Graphical constraints reinterpreted as 1..1(requires) and 0..1 (mutex) hyperarcs

Figure 2 is an attempt to graphically express these reinterpretations. While the left-hand side of Figure 2 is visually more expressive and tools can continue using that representation, the (hidden) hypergraph model (the right-hand side of Figure 2) is more easily handled by the validation/configuration algorithms.

2.2 Formal definition

A multiplicity value $mv$ is a pair of integers $mv=(min,max)$ with $min \geq 0$, $max > 0$ and $min \leq max$. We denote by $M$ the set of all possible multiplicity values, $M \subset \mathbb{N} \times \mathbb{N}$.

A Feature Diagram is an acyclic $F$-hypergraph $F= (N, E, r, \delta)$ where
- $N$ is its set of nodes (or features)
- $E=\{e_1, e_2, ..., e_m\}$, with $e_i \subseteq N$ for $i=1, ..., m$, is the set of decomposition $F$-arcs; $q$ is the cardinality of the head of $e_i$, $q=|h(e_i)|$.
- $r \in N$ is the root of the diagram (it is the only node not contained in the head of any hyperarc of the hypergraph, i.e it is the only node whose Backward Star is $\emptyset$ [8]): $R$ is the root set of the hypergraph, $R \subseteq N \land R=\{r\}$. $BS(r)=\emptyset \land BS(n) \neq \emptyset \land n \in N \land R$
- $\delta: E \rightarrow M$ assigns each $F$-arc $e$ with a multiplicity $(min, max)$, $max \leq q=|h(e)|$

A particular type of Feature Diagram is the Feature Tree. If each node has no more than one parent, then the generic graph structure is a hypertree:
- A Feature Tree FT is a Feature Diagram, such that each node has at most one entering hyperarc (root $r$ has none): $|BS(n)| = 1 \land n \in N$. $n \neq r$

Two extensions are possible. We introduce separately typed features and constraints, but the integration of the two definitions is straightforward. Constraints are introduced as additional hyperarcs with multiplicity $1..n$ (requires) or $0..1$ (mutex).

Given a Feature Diagram $F=(N,E,r,\delta)$, a Constrained Feature Diagram is a Feature Diagram $C_{\delta'}(N, E', r, \delta')$ where
- $E' = E \cup E_r \cup E_m$ and $\delta' = \delta \cup \delta_r \cup \delta_m$
\[ E_r = \{ r_1, r_2, \ldots, r_k \}, \text{ with } r_i \subseteq N \text{ for } i = 1, \ldots, k, \text{ is the set of requires constraints } F\text{-arcs}; \text{ in general } |h(r_i)| \geq 1 \]

- \( \delta_r : E_r \rightarrow M \) assigns each F-arc \( r \) with a fixed multiplicity \( \min = \max = |h(r)|. \) In particular, if the requires constraint involves two nodes, multiplicity is 1..1.

- \( E_m = \{ m_1, m_2, \ldots, m_l \}, \text{ with } m_i \subseteq N \text{ for } i = 1, \ldots, l, \text{ is the set of mutex constraints } F\text{-arcs}; \text{ in general } |h(m_i)| \geq 2 \text{ and } t(m_i) = r \text{ is the root.} \)

- \( \delta_m : E_m \rightarrow M \) assigns each F-arc \( m \) with a fixed multiplicity 0..1.

Finally, typed features allow a type (default type is NONE, others are classical predefined types such as INTEGER, BOOLEAN, STRING, etc.) to be assigned to each leaf (any feature not contained in the tail of any hyperarc, i.e. whose Forward Star is \( \emptyset \) [8]). As there is no consensus about this concept in the literature, we treat it as an optional extension.

Given a Feature Diagram \( F=(N,E,r,\delta) \), a Typed Feature Diagram is a Feature Diagram \( TF= (N, E, P, r, \delta, \tau) \) where:

- \( P \) is a set of types \( P=\{INTEGER, REAL, BOOLEAN, STRING, NONE\} \)
- \( \tau : L \rightarrow P \) assigns each leaf with a type value, \( L \subset N \land |FS(l)| = 0 \forall l \in L. \)

### 2.3 Discussion

To show the equivalence with previous FD definitions [19], a simple transformation of our Feature Diagram into a DAG can be considered: a) each hyperarc with \( h(e)=1 \) is transformed into an AND or OPTIONAL decomposition (the 1..1 or 0..1 multiplicity is assigned to the child feature); and b) each hyperarc with \( h(e) > 1 \) is transformed into a Grouped decomposition, connecting the parent and child nodes (the original multiplicity is assigned to the decomposition). The result is a constraintless DAG with multiplicity expressions or VFD, using Schobbens terminology [19]. As in [19], VFD is proved to be powerful enough to represent the rest of the FD variants, and our definition can equally generate any FD variant. As pointed out by Schobbens, expressiveness, succinctness and non-redundancy are key points. Most variants of Feature Diagram have two ways of expressing multiplicity: group multiplicity and feature multiplicity. For instance, feature multiplicity (an optional 0..1 or mandatory 1..1 feature) could be combined with an OR group, making it clear that some of the grouped features are always selected and the others are purely optional (see CreditCard in Figure 3a). This has a meaning: the group semantic indicates that the (in fact) mandatory feature is closely related to the rest of the optional features. The problem is that this possibility opens the door to unnecessary redundancies, or inconsistencies, allowing situations like Registered (optional as feature, mandatory as decomposition) in Figure 3a. The normalization of the diagrams, using only the decomposition based multiplicity is preferred, as inconsistency and redundancy are impossible, maintaining sufficient expressivity.
3 Configuration of a Feature Diagram

A (partial) configuration of a Feature Diagram is a sub-set of the original Feature Diagram where the variability is (partially) removed. In general, a manual process of node selection is carried out, obeying the constraints expressed in the Diagram. Some of these constraints are implicitly imposed by the diagram structure. Defining mandatory (non-mandatory, respectively) decompositions as decompositions where the minimum multiplicity is equal (less, respectively) than the number of its children, the following rules apply:

Rule 1. The root feature and all the features connected with the root feature through mandatory decompositions are intrinsically present in any configuration.

Rule 2. A feature connected with a selected feature through mandatory decompositions must be selected.

Rule 3. A non-mandatory feature can be selected only if at least one of its parents is selected.

Rule 4. If a feature is present, the final number $k$ of features selected as children of its decomposition must be between the minimum and maximum of the original hyperarc multiplicity: $\min \leq k \leq \max$. (Clearly, the number of children of the decomposition must be identical to the minimum and maximum of the hyperarc multiplicity for all the hyperarcs present in the configuration, i.e., no multiplicity is needed for the configuration hyperarcs.)

Other groups of constraints, when used (in a Constrained Feature Diagram), are imposed by the requires and mutex relationships:

Rule 5. Requires constraints mean that, for each feature in the configuration, all the elements required by it must also be present. In the hypergraph representation, this is equivalent to a mandatory decomposition (the two first rules apply).

Rule 6. Mutex constraints over a set of features mean that, if an involved feature is present in the configuration, the others must be absent. In the hypergraph
representation, this is equivalent to a non-mandatory decomposition with maximum multiplicity equal to 1 (the third and fourth rules apply).

Consequently, the configuration procedure can be applied uniformly to the constrained hypergraph, instead of dividing it into two phases or transforming the feature tree (or graph) into a set of propositional formulas, as proposed in the literature [16]. All the above rules can be reformulated in a more comprehensive way:

In a valid configuration, defined as a subset of features of one FD, the root feature must be present and the rest of features must satisfy two properties:

1. At least one of its *structural* parents (tail of an hyperarc not representing a mutex/requires constraint) must also be present
2. For each hyperarc whose tail is the considered feature, with min..max multiplicity, at least min and at most max children features of the head of the hyperarc must be also present in the configuration.

These properties are enough to accomplish the rules 1 to 6. The first property guarantees rule 3. For the rest of rules, considering the combination of number of children, and the min and max values of the hyperarc multiplicity, we have:

- If the number of children is 1:
  - If min=1 and max=1, this is a mandatory feature that must be present if its parent is present (rules 1 and 2); or it represents a requires constraint and the child must be present too (rule 5).
  - If min=0 and max=1, this is an optional feature that can be present or not if its parent is present;

- If the number of children is nc, being nc greater than 1:
  - If min=1 and max=1, this is an X-OR alternative feature group and one and only one of the k children features must be present if its parent is present.
  - If min=0 and max=1, this is an optional alternative feature group, and the children features can be present or not if its parent is present; or it represents a mutex constraint and at most only one of the children can be present (rule 6).
  - If 1 ≤ min < max ≤ nc, this is an OR alternative feature group and k children features must be present if its parent is present, being k a number min ≤ k ≤ max.
  - In general, if 0 ≤ min < max ≤ nc, k of the children features must be present if its parent is present, being k a number min ≤ k ≤ max.

### 3.1 Formal Definition of Valid Configuration

A *Valid Configuration* G = (NG, EG, r) is a sub-hypergraph of a (Constrained) Feature Diagram F = (N, E, r, δ) where:

- NG is a subset of nodes of N: NG ⊆ N
- EG is a set of hyperarcs: EG = {eG | ∃ e ∈ E ∧ t(eG) = t(e) ∧ h(eG) ⊆ h(e)}
- The root is present: r ∈ NG
• For each node in the configuration, at least one of its parents is present:
\[ \forall n \in N_G \ n \neq r \ \exists e \in E \ \land \ \text{type}(e) \neq \text{constraint} \ \land \ n \in h(e) \ \land \ t(e) \in N_G \]

• For each node in the configuration, at least min and at most max children features of each original associated hyperarc are present. Denoting \( h'(e) \) as the head of any hyperarc \( e \in E_G \) in the configuration \( G \), the head of \( e \) in \( G \) is a subset of the original head of \( e \) in \( F \):
\[ \forall e \in E_G \ |h'(e)| > 0 \ \land \ h'(e) \subseteq h(e) \]
\[ \forall n \in N_G \ \land \ e \in E_G \ \land \ t(e) = n \Rightarrow \text{min}(e) \leq |h'(e)| \leq \text{max}(e) \]

Considering the semantics of the FD is expressed by the FD configuration [19] we can say that an FD is valid if at least one configuration can be derived from it and if each feature of the FD is present in at least one configuration (no dead features). The trivial cases are one FD with only a feature (the root itself) or with only mandatory features (no variability at all, only a valid configuration). For the useful cases, to validate one FD, it is enough to prove that each feature is present in at least one valid configuration. The next Subsection gives a procedure to find a valid configuration of one FD, given a set of selected features. The repeated application of that procedure to each feature of the FD would serve to prove the validity of the FD itself.

3.2 Configuration Procedure of a Feature Diagram
The definition of Configuration guides the characterization of the Configure procedure. An obvious pre-condition is that the Feature Diagram is correct and has no inconsistencies.

Once the application engineer has expressed his/her preferences by selecting a set of features and, assuming they are compatible (i.e., there are no mutex or multiplicity conflicts between them), we can recognize a usual problem: it is possible that feature groups with no children selected remain undefined (hyperarcs with \( |h'(e)| = m \) and multiplicity \( 1..n < m \)). There are at least two ways in which the configuration process can be dealt with: a) finding the (probably ordered) set of all valid configurations that fulfill the defined selection; and b) guiding the engineer until a unique valid configuration is found. The first option is a complete but computationally costly solution. The second is more realistic, but it remains largely a manual process, accomplished with FD tools. Staged configuration [7] is a classical approach for solving this problem in several steps.

We find using a topological order in the set of features included in the head of each hyperarc useful for facilitating the process. For F-graphs, such node preordering can be accomplished by the F-Acyclic procedure [8]. This option implies that the domain engineer has assigned a preference order to each group of features (alternatively the “weight” of the node plus its mandatory descendants could be automatically calculated and assigned to the features). The aim is to have a (set of) default feature(s) when there is not an explicit decision. An example can clarify the idea: in an e-commerce product line, credit card payment is more frequent than check or phone based payments and, in consequence, if the application engineer does not explicitly decide to change the payment method, credit payment will be selected by default.
In any case the possible configurations can be generated in two steps that try to achieve properties 1 and 2 respectively:

1. The partial configurations that include the selected features are found. This step is deterministic in the sense that one path from each selected feature to root must be included, and (recursively) mandatory descendants of each feature in the configuration must be present. If a feature must be incorporated but it is incompatible with the already present features, the partial configuration is discarded. The resulting partial configurations can be communicated to the application engineer or used in step 2.

2. Each resulting partial configuration must be completed using a second procedure that must find the most “economical” final configuration of the FD. For each undefined hyperarc (the number of selected features of its head is less than the min multiplicity and the hyperarc tail is present) the default feature descendant(s) are added to reach the min multiplicity. Again each added feature must be compatible with the partial configuration or that possibility is discarded.

The first step is a variation of the hypergraph \textit{BVisit} algorithm. The second one can be a refinement of the \textit{FVisit} algorithm, both described in [8]. Thus, given a (Constrained) Feature Diagram \(F=(N, E, r, \delta)\), and \(U\) an identified (selected manually) subset of compatible nodes of \(N\): \(U \subseteq N\), a set of valid configurations \(G=(N_G, E_G, r)\) is obtained in two steps.

### 3.3 Step 1 Implementation

We say that one hyperarc \(e \in F\) is a \textit{parent} hyperarc of node \(u\), if the head of the hyperarc contains \(u\), \(u \in \text{h}(e)\), and it is \textit{structural} (i.e., not a constraint type hyperarc). Value \text{used}(e) gives the number of children of \(e\) included in the configuration so far. A selected feature requires that, at least, a parent hyperarc is included in the configuration. We start always from an initial configuration of \(F\), \(G_0=(N_0, E_0)\) where \(N_0\) includes only the root node of \(F\) and \(E_0\) has zero selected hyperarcs.

For \(U\) and \(G\), being \(G\) a (partial) configuration of \(F\), a procedure \textit{configure}(\(U, G\)), adapted from \textit{BVisit}(\(F, n\)) [8] is applied (See Procedure 1 basic scheme).

For each node \(u \in U\), \(u\) is selected and removed. For each hyperarc \(e\) \textit{parent} of \(u\), (i.e., requires or mutex arcs are not selected), and if it is a \textit{valid parent hyperarc} (the hyperarc \text{used}(e) value is less than the hyperarc max multiplicity), the configuration process continues: the tail of \(e\) is added to the list of selected features and a new recursive execution of the procedure 1 is launched, starting from the current configuration. If there are no \textit{valid parents hyperarcs} for a node, the procedure execution aborts (though other branches can continue).

Clearly the complexity of the procedure depends on the number of nodes with more than one parent. In the implemented version or the procedure, each time one node is added, two operations are used for efficiency reasons: (1) its mandatory
descendants are also added to the selected nodes, using an auxiliary procedure; (2) the existence of conflicts in the resulting partial configuration is tested using an auxiliary compatible($G$) function (returns false if used(e)$=$max(e) in any hyperarc of the configuration) to discard illegal configurations as soon as possible.

Procedure 1 transforms the initial $U$ and $G_0=(N_0, E_0)$ into one set of partial configurations $G'_i=(N_i, E_i)$, all of them compatible with the expressed requirements. For each resulting modified $G'_i$, property 1 holds at this point, that is, each feature has at least one structural parent (and no conflicts are present). Property 2 must be accomplished in a second step.

```
procedure configure(U, G)
  While U ≠ ∅ do
    --select and remove u ∈ U, mark u in G as selected
    u:= U.first; U := U – {u};
    select(u); --mark node as selected
    valid_parents:=0;
    For each e ∈ BS(u) and type(e) ≠ “constraint”
      --selects hyperarc if possible (not used max times)
      If used(e) < max(e)
        valid_parents:= valid_parents+1;
        incrementUsed(e);
        If not selected(Tail(e))
          select(Tail(e));
      End if
      U := U – Tail(e);
    configure(U, G); --recursive call
  End for each
  G:=void; --finished path
  If valid_parents = 0
    return; --no valid configuration **invalid branch
  End if
  End while
  If G ≠ void
    complete(G); --partial configuration to be completed
  End if
End procedure
```

**Procedure 1.** Generation of the staged (partial) configurations

3.4 Step 2 Implementation

Procedure complete($G'_i$), inspired on $FVisit$ [8] is applied to each modified partial configuration $G'_i$. Original $FVisit$ finds all nodes connected to root $r$ and returns a set of paths connecting them to $r$. It must be adapted in order to limit the number of nodes of $h(e)$ to be examined. Note that for each feature group, only min features must be selected, being (min, max) their associated multiplicity value. The greedy algorithm can be improved if we consider first the default features. As the algorithm
is recursive, the first termination reaches the configuration where all the choices are the default feature(s). Only when that configuration is illegal the algorithm continues searching for the next valid configurations. The procedure selects nodes to be included in the final configuration hypergraph $G_i=(N_{Gi}, E_{Gi})$.

As in procedure 1, in the optimized implemented version, the mandatory descendants of each new node selected are also added to the selected nodes, using an auxiliary procedure (not shown in the basic scheme of Procedure 2). If new node(s) with no parents are added to the configuration, a new call to the $configure(U,G_i)$ procedure is necessary to find their parents before completion ($U$ includes the new nodes without parents).

```
procedure complete(G)
    create W = {e_1, e_2, ... e_n} where selected(tail(e_i)) = true;
    While W ≠ ∅ do
        -- select and remove $e \in E$
        $e := W$.first; W := W - {e};
        For each node $n \in head(e)$
        -- select node and check if there is no conflicts
        If |head(e)| < min(e)
            If selected(n) = false
                select(n);
                incrementUsed(e);
                If compatible(G)
                    complete(G); -- recursive call
            Else
                G := void; -- no valid configuration
                return;
            End if
        End if
    End for each
    G := void; -- finished path
    If |head(e)| < min(e)
        return; -- no valid configuration
    End if
End while
If G ≠ void
    display(G); -- a final configuration reached
End if
End procedure
```

Procedure 2. Generation of a complete configuration from a partial configuration

The minimal set of features that are present in a configuration if no features are selected (built applying recursively procedures $configure/complete$ to the root feature and selecting first min children in each hyperarc) constitute the default feature configuration. The minimal set of features that are present in all the possible configurations (built applying only procedure $configure$ to the root feature and
selecting only mandatory/requires features where \( \min(e) = |\text{head}(e)| \) in each considered hyperarc) constitute the core features. The base package of the SPL architecture must give a design solution to these core features.

A basic implementation in Java has been coded to test the procedures and to estimate the time needed to reach the partial and final configurations. The first FD used to collect representative values has been the graph product-line problem proposed in [15] for evaluating product-line methodologies. The time elapsed to find partial and final configurations is less than one second in a modern PC. Thus, the use of hypergraphs as a practical tool with standard FDs is viable. The scalability of the proposal with FD with hundreds of features and constraint is under study.

Appendix A shows the output of the procedures application to the mentioned example. The Table 1 shows some of the generated configurations.

Table 1. Generated configurations for the example FD proposed in [15]

<table>
<thead>
<tr>
<th>Selected nodes</th>
<th>Elapsed time (ms)</th>
<th>Generated configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>{10, 11}</td>
<td>31</td>
<td>{0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 6(6.weighted); 8(8.bfs); 10(10.number); 11(11.connected); }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 6(6.weighted); 9(9.dfs); 10(10.number); 11(11.connected); }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 7(7.unweighted); 8(8.bfs); 10(10.number); 11(11.connected); }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 6(6.weighted); 9(9.dfs); 10(10.number); 11(11.connected); 14(14.mstPrim); }</td>
</tr>
<tr>
<td>{10, 11, 14}</td>
<td>36</td>
<td>{0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 6(6.weighted); 8(8.bfs); 10(10.number); 11(11.connected); 14(14.mstPrim); }</td>
</tr>
<tr>
<td>{10, 11, 14, 15}</td>
<td>3</td>
<td>[Incompatible nodes...]</td>
</tr>
<tr>
<td>{}</td>
<td>18</td>
<td>{0(0.root-gpl); 1(1.gtp); 3(3.alg); 4(4.directed); 6(6.weighted); }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{0(0.root-gpl); 1(1.gtp); 3(3.alg); 4(4.directed); 7(7.unweighted); }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{0(0.root-gpl); 1(1.gtp); 3(3.alg); 5(5.undirected); 6(6.weighted); }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{0(0.root-gpl); 1(1.gtp); 3(3.alg); 5(5.undirected); 7(7.unweighted); }</td>
</tr>
</tbody>
</table>
4 Feature Meta-model

One of the advantages of the definition of Feature Diagrams as F-hypergraphs is that we have only two types of elements, features and decompositions, instead of introducing an additional element (grouped features) to complete the semantics. In consequence, the definition and implementation (as CASE tools) of the meta-model is easier. The proposal is modular, allowing several versions, from the simplest Tree based meta-model to the complete F-hypergraph/constrained/typed meta-model. The definition style uses the package merge mechanism and is the same that the UML2 meta-model uses extensively in the OMG documentation. This approach allows all the variants of the feature diagrams mentioned in Sections 1 and 5 to be covered (figure 4).

As an example, the details of the base package are shown in Figure 5. A FeatureDiagram has a Root (feature), a set of zero or more (non-root) Features, and a set of HyperArcs (in this case Decompositions). Each Decomposition connects a parent Node (Root or Feature) with one or more child Features. As multiplicity of children meta-association indicates, a Feature can only be child of one Decomposition.
(and indirectly of a parent feature). This is a constraint that makes the structure into a tree with a root that has no parents. Decomposition has an associated *MultiplicityElement* that must conform to the associated OCL constraint: maximum value (upper) must be less than or equal to the number of children of the Decomposition. To convert a tree based meta-model into the general F-hypergraph version, we need to merge the package DAG. Other possibilities are the Typed or Constraints packages.

![Feature Diagram](image)

**Fig. 5** Detail of the basic package of the proposed extensible Feature Meta-model

The basic meta-model of the Feature Configuration (rarely used) is the same of Figure 10 but with the invariant \(0 \leq \min = \max = \text{self.children-> size}\), taken into account that the possibility of election among children features must be null.

To convert a tree based meta-model into the general F-hypergraph version, we need to merge the package DAG. Other possibilities are the Typed or Constraints packages. As an example the combination of the four packages, respecting the UML merge rules results in the meta-model of Figure 6.
Fig. 2 Final Constrained Typed DAG version of Feature Meta-model
To implement the meta-model, we have used GMF\(^1\). The Eclipse Graphical Modeling Framework (GMF) provides a generative component and runtime infrastructure for developing graphical editors based on EMF and GEF. A plain implementation of the meta-model has been defined. The visual syntax of the Features and Root are rectangles, while the Decomposition graphical representation is a circle with multiplicity details. The Tree/DAG variants require a different multiplicity value but visually are similar. The type is easy to add as an attribute of Feature. The graphical constraints are simple arrows.

\(^1\) http://www.eclipse.org/modeling/gmf/

5 Transformation of Feature Models

Feature Diagrams have associated design models (generally expressed by UML models, including class, use case, and interaction diagrams). We use the UML package merge mechanism to preserve the traceability form feature to UML models, as explained in [13]. The proposed meta-model (and the consequent tool availability) opens the door to the definition of enhanced versions of the feature model to UML transformation presented in [13]. We propose to refine the original version (based on feature trees) into a more general version. We face the problem in two phases:

- The simplest situation occurs when the hypergraph is a hypertree (and then the transformation is trivial)
- The general case when a F-hypergraph is considered
Feature Tree transformation

If the Feature Diagram is a Feature Tree, the strategy consist of that each variability point detected in the feature model must originate a package that will be combined, or not, in product development time, according to the selected configuration [13].

If we define a UML Tree Package Model $PM = (P, M)$, where $P$ is the a set of packages and $M$ the set of ordered pairs of packages (representing merge dependencies between them, $p1$ requires $p2$): $m(p1, p2) \in M$, $p1, p2 \in P \land p1 \Rightarrow p2$.

Definition of Feature Tree Transformation Operation: $FT \rightarrow TPM$

Given a Feature Tree $h = (N, E, r, \delta)$, a UML Package Model $PM = (P, M)$ is created applying the following rules:

- The root $r$ generate the Base package $P=\{\text{Base}\} \land pp=\text{Base}$
- Each feature (recursively) connected by an optional hyperarc decomposition to a previous considered feature of $N$ (including root) generates a new package and a new merge dependency from the new package to the previous.

$$\forall n \in N \quad \forall e \in E \cdot (T(e) = \{n\} \land |H(e)| > \text{min}(e)) \Rightarrow$$

$$p1 \text{ is new } P = P \cup \{p1\} \land m1 \text{ is new } m1(p1, pp). M = M \cup \{m1\} \land pp=p1$$

In this type of transformation, where different meta-models are implied, the MDE approach is a better approach. We have previously implemented this transformation using the Czarnecki meta-model. Using the new meta-model, the transformation is easier. Figure 8 shows the QTV based definition of the transformation.

In practical terms we must consider the multiplicity of the parent decomposition. If the minimum is equal to the number of children of the decomposition (1..1, for instance, when the number of children is exactly one) the features are non optional and the related design elements must be incorporated to the existing package. If the minimum is less to the number of children (1..1, when the number of children is more than one; 0..1; 0..2; 1..2; etc.) the feature is optional and the design elements are described in a new package, merged with the existing package.

Being a tree, the transformation can be implemented by a XML style sheet and involves:

a) Transform the Feature model into a UML model.

b) Transform the RootFeature into a root Package

c) Transform each optional Feature (i.e., with multiplicity minimum less than the number of children of the decomposition from which it is part) into a package merged with the previous package.

d) Ignore the mandatory Features (i.e., with minimum multiplicity equal to the number of children), simply passing to nodes in the next level.

An example of simple application is shown in Figure 9.
Fig. 8 Using QVT to define the proposed transformation
Feature Diagram transformation

If the Feature Diagram is a general hypergraph the transformation is more complex as a feature can have contradictory properties. The chosen strategy implies that each package with more than one parent in the feature model originates always a package. An optional decomposition is treated as previously and generates a package (if not created before) and a merge dependency. A non optional feature generates a package (if not created before) and one import dependency with the base package that in this case is the source of the relationship, while the new package is the target. Figure 10 shows an example of this strategy. The cost we assume is that the structure is in general not optimized. Considering the CreditCard package, we can argue that if is included in Payment, the Guest package always has access to the content of CreditCard and we could remove it, getting exactly the same model of Figure 9.
If we define a UML Package Model pm = (P, M, I), where P is the set of packages, M the set of ordered pairs of packages (representing merge dependencies between them, p1 requires p2): m(p1, p2) ∈ M. p1, p2 ∈ P ∧ p1 ⇒ p2 and I the set of ordered pairs of packages (representing import dependencies between them, p1 requires p2): i(p1, p2) ∈ M. p1, p2 ∈ P ∧ p1 ⇒ p2

Definition of Feature Diagram Transformation Operation: FD → PM

Given a Feature Diagram FD h = (N, E, r, δ), a UML Package Model pm = (P, M, I) is created applying the following rules:

- The root r generate the Base package P={Base} ∧ pp=Base
- Each feature (recursively) connected by an optional hyperarc decomposition to a previous considered feature of N (including root) generates a new package and a new merge dependency from the new package to the previous.
  ∀ n ∈ N ∀ e ∈ E . (T(e)={n} ∧ |H(e)|>min(e)) =>
  p1 is new P = P ∪ {p1} ∧ m is new m(p1, pp). M = M ∪ {m} ∧ pp=p1
- Each feature connected by a non optional hyperarc decomposition to a previous considered feature of N (including root) and with more than one parent decomposition generates a new package and a new import dependency from the previous package to the new.
  ∀ n ∈ N ∀ e ∈ E . (T(e)={n} ∧ |H(e)|=min(e) ∧ |BS(n)| > 1) =>
  p2 is new P = P ∪ {p2} ∧ i is new i(pp, p2). I = I ∪ {i} ∧ pp=p2

The general hypergraph based meta-model transformation is described with QVT in Figure 11. As in the previous transformation, we consider multiplicity but also the number of parents. If the feature is optional, the design elements are described in a new package, merged with the existing package. If the features are non optional and the number of parents is exactly one, the related design elements must be incorporated to the existing package. But if the features are non optional and the number of parents is greater than one, the design elements are described in a new package, imported by the existing package. A minor problem to manage is the fact that the new package can exist as result of a previous feature transformation. Therefore we have defined a total of eight QVT transformations.
Using QVT to define the proposed transformation

Being a single rooted graph, the transformation can be implemented by a XML style sheet and involves:

e) Transform the Feature model into a UML model.

f) Transform the RootFeature into a root Package

g) Transform each optional Feature (i.e., with multiplicity minimum less than the number of children of the decomposition from which it is part) into a package merged with the previous package.

h) If the number of parents of the feature are exactly one, ignore the non optional Features (i.e., with minimum multiplicity equal to the number of children), simply passing to nodes in the next level.

i) If the number of parents of the feature are greater than one, transform each non optional feature into a package imported by the previous package.

6 Related work

Starting with the original FODA proposal [12], several variants of feature diagrams have been proposed: FORM [11] is an extension where feature diagrams are single-rooted directed acyclic graphs (DAG) instead of simple trees. FeatureRSEB [10] also uses DAGs and changes the visual syntax, including a graphical representation for the constraints requires and mutex. Other authors, such as Czarnecki et al. [5,6] and Batory [1], continue to use trees as the main structure (however Czarnecki et al. add OR decomposition, graphical constraints, and distinguish between group and feature cardinalities). Riebisch et al. [18] replace AND, X-OR, and OR by multiplicities combined with mandatory and optional edges. Cechticky et al. proposed a notation
without solitary features in an attempt to reduce the number of redundant representations: a group with one grouped feature is used instead [4].

A detailed comparison of all these variants has been done by Schobbens et al. in [19]. The authors use a parameterized formal definition of the feature diagram, obtaining a framework useful for comparing and classifying all the variants, proving how the diverse options can be equivalent. Some recent works are devoted to the validation of feature models, mainly based on propositional formulas [1] or constraint solvers [2]. Mendoça et al. use a two stage analysis to validate the models [16]. The advantage of using hypergraphs is the remarkable simplification of the supporting model. Instead of transforming FDs into a set of formulas to find inconsistencies or configure the final product, the algorithms can be used directly on the constrained hypergraphs, using a unique formalism. Modeling and transformation tools are easier to define and implement as a coherent and extensible set.

6 Conclusions and future work

In this article, we have used F-hypergraphs to define the semantics of feature diagrams and their configuration. Once the formal definition is stated, the construction of an extensible feature meta-model has been dealt with. The algebraic definition directly yields the invariants of the meta-model, establishing a firm foundation. The advantages of simplicity and extensibility have made it possible to build a set of modeling tools compatible with the different flavors of FDs.

As part of our industrial oriented work, we implemented a Feature Modeling Tool (FMT) as a template integrated with the Microsoft Visual Studio IDE. The meta-model we used was based on constrained trees, validation is external, and configuration uses a staged approach. Work in progress on FMT includes the incorporation of the extensible meta-model and the implementation of the algorithms. As the tool was originally built using DSL tools and C#, the meta-model enhancement and the implementation of the algorithms are straightforward.

References

Appendix A. Example of FD (GMF model)

<?xml version="1.0" encoding="UTF-8"?>
<hyper:FeatureDiagram xmlns:xmi="http://www.omg.org/XMI"
xmlns:hyp="http://www.eclipse.org/hypergraph/xmi"
xmlns:hyper="http://www.eclipse.org/hypergraph/hyper.ecore"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <arcs xsi:type="hyper:Decomposition" children="/@nodes.1" cardinality="1..1" parent="/@nodes.0"/>
  <arcs xsi:type="hyper:Decomposition" children="/@nodes.3" cardinality="0..1" min="0" parent="/@nodes.0"/>
  <arcs xsi:type="hyper:Decomposition" children="/@nodes.6" cardinality="1..1" parent="/@nodes.0"/>
  <arcs xsi:type="hyper:Decomposition" children="/@nodes.1 //@nodes.2 //@nodes.5" cardinality="1..1" parent="/@nodes.3"/>
  <arcs xsi:type="hyper:Decomposition" children="/@nodes.4 //@nodes.9" cardinality="1..1" parent="/@nodes.3"/>
  <arcs xsi:type="hyper:Decomposition" children="/@nodes.7 //@nodes.8" cardinality="1..1" parent="/@nodes.1"/>
  <arcs xsi:type="hyper:Decomposition" children="/@nodes.10 //@nodes.11 //@nodes.15 //@nodes.14 //@nodes.12 //@nodes.16 //@nodes.13" cardinality="0..6" min="0" max="6" parent="/@nodes.6"/>
  <arcs xsi:type="hyper:Requirements" children="/@nodes.3" cardinality="1..1" source_r="/@nodes.10"/>
  <arcs xsi:type="hyper:Mutex" children="/@nodes.12 //@nodes.16" cardinality="0..1" min="0"/>
  <arcs xsi:type="hyper:Requirements" children="/@nodes.1 //@nodes.3" cardinality="2..2" min="2" max="2" source_r="/@nodes.15"/>
  <arcs xsi:type="hyper:Requirements" children="/@nodes.4 //@nodes.9" cardinality="2..2" min="2" max="2" source_r="/@nodes.15"/>
  <arcs xsi:type="hyper:Requirements" children="/@nodes.7 //@nodes.8" cardinality="2..2" min="2" max="2" source_r="/@nodes.15"/>
  <arcs xsi:type="hyper:Requirements" children="/@nodes.10 //@nodes.11 //@nodes.15 //@nodes.14 //@nodes.12 //@nodes.16 //@nodes.13" cardinality="1..1" source_r="/@nodes.10" />
  <nodes xsi:type="hyper:RootFeature" name="root-gpl"/>
  <nodes xsi:type="hyper:Feature" name="gtp"/>
  <nodes xsi:type="hyper:Feature" name="directed"/>
  <nodes xsi:type="hyper:Feature" name="src"/>
  <nodes xsi:type="hyper:Feature" name="weighted"/>
  <nodes xsi:type="hyper:Feature" name="unweighted"/>
  <nodes xsi:type="hyper:Feature" name="dfs"/>
  <nodes xsi:type="hyper:Feature" name="number"/>
  <nodes xsi:type="hyper:Feature" name="strongC"/>
  <nodes xsi:type="hyper:Feature" name="mstPrim"/>
  <nodes xsi:type="hyper:Feature" name="cycle"/>
  <nodes xsi:type="hyper:Feature" name="connected"/>
  <nodes xsi:type="hyper:Feature" name="mstKruskal"/>
</hyper:FeatureDiagram>
Appendix B. Examples of Configuration

Nodes in FD: {0(root-gpl); 1(gtp); 2(src); 3(alg); 4(directed); 5(undirected); 6(weighted); 7(unweighted); 8(bfs); 9(dfs); 10(number); 11(connected); 12(strongC); 13(cycle); 14(mstPrim); 15(mstKruskal); 16(shortest); }

Hyperarcs:
From[mult]H(To):
0 = 0[1,1]-H:0-{1,}
1 = 0[0,1]-H:0-{2,}
2 = 0[1,1]-H:0-{3,}
3 = 1[1,1]-H:0-{4,5,}
4 = 1[1,1]-H:0-{6,7,}
5 = 2[1,1]-H:0-{8,9,}
6 = 3[0,6]-H:0-{10,11,12,13,14,15,16,}
7 = 0[0,1]-H:0-{14,15,}
8 = 10[1,1]-H:0-{2,}
9 = 11[2,2]-H:0-{2,5,}
10 = 12[2,2]-H:0-{4,9,}
11 = 13[1,1]-H:0-{9,}
12 = 14[2,2]-H:0-{5,6,}
13 = 15[2,2]-H:0-{5,6,}
14 = 16[2,2]-H:0-{4,6,}
B1.
Output of procedure 1 and 2 for the mentioned example of [15] and nodes 10(10.number) and 11(11.connected) selected.

Selected Nodes = {10, 11}

staged configuration!! to COMPLETE

Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 10(10.number); 11(11.connected); }

Hyperarcs:
0 = 0[1,1]-H:1-{1,}
1 = 0[1,1]-H:1-{2,}
2 = 0[1,1]-H:1-{3,}
3 = 1[1,1]-H:1-{4,5,}
4 = 1[1,1]-H:0-{6,7,}
5 = 2[1,1]-H:0-{8,9,}
6 = 3[2,6]-H:2-{10,11,12,13,14,15,16,}
7 = 0[0,1]-H:0-{14,15,}
8 = 10[1,1]-H:1-{2,}
9 = 11[2,2]-H:2-{2,5,}
10 = 12[2,2]-H:0-{4,9,}
11 = 13[1,1]-H:0-{9,}
12 = 14[2,2]-H:0-{5,6,}
13 = 15[2,2]-H:0-{5,6,}
14 = 16[2,2]-H:0-{4,6,}

node 6
*** recursive call
node 8
*** recursive call
*****COMPLETED

Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 6(6.weighted); 8(8.bfs); 10(10.number); 11(11.connected); }

node 9
*** recursive call
*****COMPLETED

Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 6(6.weighted); 9(9.dfs); 10(10.number); 11(11.connected); }

node 7
*** recursive call
node 8
*** recursive call
*****COMPLETED

Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 7(7.unweighted); 8(8.bfs); 10(10.number); 11(11.connected); }

node 9
*** recursive call
*****COMPLETED

Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 7(7.unweighted); 9(9.dfs); 10(10.number); 11(11.connected); }

Elapsed time: 31 ms
B2.
Output of procedure 1 and 2 for the mentioned example of [15] and nodes
10(10.number), 11(11.connected), and 14(14.mstPrim) selected.

Selected Nodes = {10, 11, 14}

staged configuration!! to COMPLETE
Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 6(6.weighted); 10(10.number); 11(11.connected); 14(14.mstPrim); }

Hyperarcs:
0 = 0[1,1]-H:1-{
1 = 0[1,1]-H:1-{
2 = 0[1,1]-H:1-{
3 = 1[1,1]-H:1-{
4 = 1[1,1]-H:1-{
5 = 2[1,1]-H:0-{
6 = 3[3,6]-H:3-{
7 = 0[0,1]-H:0-{
8 = 10[1,1]-H:1-{
9 = 11[2,2]-H:2-{
10 = 12[2,2]-H:0-{
11 = 13[1,1]-H:0-{
12 = 14[2,2]-H:2-{
13 = 15[2,2]-H:0-{
14 = 16[2,2]-H:0-{

node 8
*** recursive call
Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 6(6.weighted); 8(8.bfs); 10(10.number); 11(11.connected); 14(14.mstPrim); }

node 9
*** recursive call
Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 6(6.weighted); 9(9.dfs); 10(10.number); 11(11.connected); 14(14.mstPrim); }

Elapsed time: 36ms

B3.
Output of procedure 1 and 2 for the mentioned example of [15] and nodes
10(10.number), 11(11.connected), 14(14.mstPrim), and 15(mstKruskal) selected. (**14 and 15 are mutually exclusive**)

Selected Nodes = {10, 11, 14, 15}

{incompatible nodes...}

Elapsed time: 3ms
B4.
Output of procedure 1 and 2 for the mentioned example of [15] and no
nodes selected (**only root is in the initial configuration**).
Selected Nodes = {}  

staged configuration!! to COMPLETE
Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 3(3.alg); }

Hyperarcs:
0 = 0[1,1]-H:1-{1,}
1 = 0[0,1]-H:0-{2,}
2 = 0[1,1]-H:1-{3,}
3 = 1[1,1]-H:0-{4,5,}
4 = 1[1,1]-H:0-{6,7,}
5 = 2[1,1]-H:0-{8,9,}
6 = 3[0,6]-H:0-{10,11,12,13,14,15,16,}
7 = 0[0,1]-H:0-{14,15,}
8 = 10[1,1]-H:0-{2,}
9 = 11[2,2]-H:0-{2,5,}
10 = 12[2,2]-H:0-{4,9,}
11 = 13[1,1]-H:0-{9,}
12 = 14[2,2]-H:0-{5,6,}
13 = 15[2,2]-H:0-{5,6,}
14 = 16[2,2]-H:0-{4,6,}

node 4  
*** recursive call
node 6  
*** recursive call
*****COMPLETED
Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 3(3.alg); 4(4.directed); 6(6.weighted); }

node 7  
*** recursive call
*****COMPLETED
Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 3(3.alg); 4(4.directed); 7(7.unweighted); }

node 5  
*** recursive call
node 6  
*** recursive call
*****COMPLETED
Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 3(3.alg); 5(5.undirected); 6(6.weighted); }

node 7  
*** recursive call
*****COMPLETED
Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 3(3.alg); 5(5.undirected); 7(7.unweighted); }

Elapsed time: 18ms
B5.
Nodes in FD: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 4(4.directed); 5(5.undirected); 6(6.weighted); 7(7.unweighted); 8(8.bfs); 9(9.dfs); 10(10.number); 11(11.connected); 12(12.strongC); 13(13.cycle); 14(14.mstPrim); 15(15.mstKruskal); 16(16.shortest); } 

Hyperarcs: 
From[mult] H{To}:
0 = 0[1,1]-H:0-{1,}
1 = 0[0,1]-H:0-{2,}
2 = 0[1,1]-H:0-{3,}
3 = 1[1,1]-H:0-{4,5,}  *****
4 = 1[1,1]-H:0-{6,7,}
5 = 2[1,1]-H:0-{7,8,9,}  *****
6 = 3[0,6]-H:0-{10,11,12,13,14,15,16,}
7 = 0[0,1]-H:0-{14,15,}
8 = 10[1,1]-H:0-{2,}
9 = 11[2,2]-H:0-{2,5,}
10 = 12[2,2]-H:0-{4,9,}
11 = 13[1,1]-H:0-{9,}
12 = 14[2,2]-H:0-{5,6,}
13 = 15[2,2]-H:0-{5,6,}
14 = 16[2,2]-H:0-{4,6,}

Output of procedure 1 and 2 for the mentioned example of [15] modified in arc 5(node 7 has two incoming hyperarcs); nodes 10(10.number) and 14(14.mstPrim) selected:

Selected Nodes = {10, 14}
staged configuration!! to COMPLETE
Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 6(6.weighted); 10(10.number); 14(14.mstPrim); }

Hyperarcs:
0 = 0[1,1]-H:1-{1,}
1 = 0[1,1]-H:1-{2,}
2 = 0[1,1]-H:1-{3,}
3 = 1[1,1]-H:1-{4,5,}
4 = 1[1,1]-H:1-{6,7,}
5 = 2[1,1]-H:0-{7,8,9,}
6 = 3[2,6]-H:2-{10,11,12,13,14,15,16,}
7 = 0[0,1]-H:0-{14,15,}
8 = 10[1,1]-H:1-{2,}
9 = 11[2,2]-H:0-{2,5,}
10 = 12[2,2]-H:0-{4,9,}
11 = 13[1,1]-H:0-{9,}
12 = 14[2,2]-H:0-{5,6,}
13 = 15[2,2]-H:0-{5,6,}
14 = 16[2,2]-H:0-{4,6,}

node 7 
***** incompatible configuration ***

node 8
*** recursive call
*****COMPLETED
Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 6(6.weighted); 8(8.bfs); 10(10.number); 14(14.mstPrim); }

node 9
### B6.

Output of procedure 1 and 2 for the mentioned example of [15] modified in arc 5 (node 7 has two incoming hyperarcs); nodes 10 (10.number) and 7 (7.unweighted) selected:

Selected Nodes = {10, 7}

staged configuration!! to COMPLETE

Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg);
7(7.unweighted); 10(10.number); }

Hyperarcs:
0 = 0[1,1]-H:1-{1,}
1 = 0[1,1]-H:1-{2,}
2 = 0[1,1]-H:1-{3,}
3 = 1[1,1]-H:0-{4,5,}
4 = 1[1,1]-H:1-{6,7,}
5 = 2[1,1]-H:0-{7,8,9,}
6 = 3[1,6]-H:1-{10,11,12,13,14,15,16,}
7 = 0[0,1]-H:0-{14,15,}
8 = 10[1,1]-H:1-{2,}
9 = 11[2,2]-H:0-14,5,}
10 = 12[2,2]-H:0-4,9,}
11 = 13[1,1]-H:0-{9,}
12 = 14[2,2]-H:0-5,6,}
13 = 15[2,2]-H:0-5,6,}
14 = 16[2,2]-H:0-4,6,}

node 4

*** recursive call
node 7
node 8

*** recursive call

*****COMPLETED

Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg);
4(4.directed); 7(7.unweighted); 8(8.bfs); 10(10.number); }

node 9

*** recursive call

*****COMPLETED

Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg);
4(4.directed); 7(7.unweighted); 9(9.dfs); 10(10.number); }

node 5

*** recursive call
node 7
node 8

*** recursive call

*****COMPLETED
Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 7(7.unweighted); 8(8.bfs); 10(10.number); }

node 9
*** recursive call
*****COMPLETED
Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 9(9.dfs); 10(10.number); }

Hyperarcs:
0 = 0[1,1]-H:1-{1,}
1 = 0[1,1]-H:1-{2,}
2 = 0[1,1]-H:1-{3,}
3 = 1[1,1]-H:0-{4,5,}
4 = 1[1,1]-H:0-{6,7,}
5 = 2[1,1]-H:1-{7,8,9,}
6 = 3[1,6]-H:1-{10,11,12,13,14,15,16,}
7 = 0[0,1]-H:0-{14,15,}
8 = 10[1,1]-H:1-{2,}
9 = 11[2,2]-H:0-{2,5,}
10 = 12[2,2]-H:0-{4,9,}
11 = 13[1,1]-H:0-{9,}
12 = 14[2,2]-H:0-{5,6,}
13 = 15[2,2]-H:0-{9,}
14 = 16[2,2]-H:0-{4,6,}

node 4
*** recursive call
node 6
*** recursive call
*****COMPLETED
Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 4(4.directed); 6(6.weighted); 7(7.unweighted); 10(10.number); }

node 5
*** recursive call
node 6
*** recursive call
*****COMPLETED
Nodes in configuration: {0(0.root-gpl); 1(1.gtp); 2(2.src); 3(3.alg); 5(5.undirected); 6(6.weighted); 7(7.unweighted); 10(10.number); }

Elapsed time: 58ms