Chapter 4: The AddAtom lattice construction algorithm

In this chapter we describe and define a concept lattice construction algorithm called AddAtom. This is done in two parts. In the first, in section 4.1, we give an informal description of the strategies used in the AddAtom lattice construction algorithm using a graph theoretic view. Then, in section 4.2, we describe the relation of the intent- and extent representative operations defined in chapter 2 to lattice construction and show that these operations have a direct relationship to the structural properties of a lattice. In section 4.3 the AddAtom algorithm is formally defined in pseudo code using a set theoretic point of view. This is followed by an example of the execution of the algorithm (section 4.4). Since the algorithm defined in section 4.3 is very inefficient as stated, section 4.6 considers efficient implementations of the algorithm derived from an efficient algorithm for determining the intent- and extent representative operations (section 4.5). The chapter concludes with a general discussion of the algorithm (section 4.7).

4.1 INFORMAL DESCRIPTION

This section is an informal discussion of the AddAtom concept lattice construction algorithm. The description incrementally builds an understanding of the algorithm by describing the various strategies used in the algorithm. This approach is taken to give the reader an intuitive understanding of lattice construction without trying to decipher the concepts of a more formal description. In the next section a formal description of the algorithm is given. Readers familiar with lattice construction may wish to skip this section.

As a starting point, an observation that can be made about the inefficient algorithm defined in the previous chapter (BruteForceEAConstruct) is that it ignores the information already contained in the lattice L_n. The algorithm computes all concepts and consider each as possibilities regardless of whether there is a likelihood of finding any new EA-formal concept or not. However by inspecting the nodes and arcs in L, the creation of a number of concepts could have been avoided (e.g. generating only combinations of attributes that actually occur in I). The process of creating arcs could also be significantly improved by using the "information already contained in the lattice". This idea of using the information already contained in the lattice is the key to the AddAtom algorithm. It is therefore worthwhile to take a closer look at the lattice before defining the algorithm in order to see how the lattice itself can be used to more efficiently construct L_{n+1}.

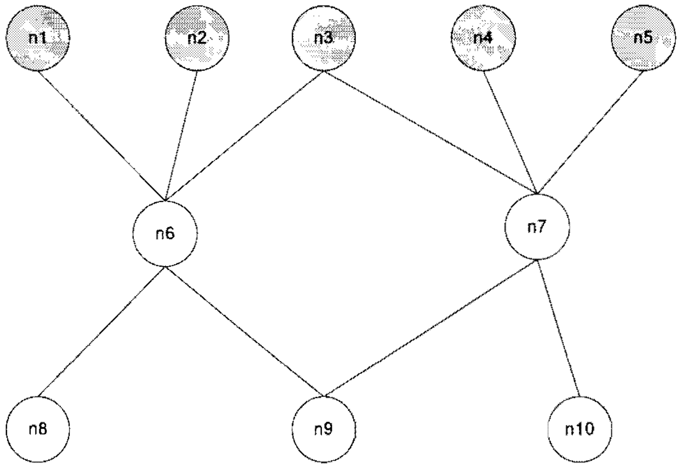

Figure 4.1: Nodes in a lattice are connected to the meet of subsets of their intents

In general, any node is always connected to the meet of some subset of the attributes in its intent. For example node n_9 in figure 4.1 has an intent of $\{n_1, n_2, n_3, n_4, n_5\}$. In this case n_9 is connected to n_6 and n_7, the meets of $\{n_1, n_2, n_3\}$ and $\{n_3, n_4, n_5\}$ respectively. Since the node itself is the meet of all the attributes in its intent it seems that if we want to insert a new object node e into the lattice, we must find the meets $m_1 \ldots m_j$ of all subsets of n's intent and connect the new node to some of these meets. However if such a meet is spanned by another meet (not the unit concept), lower down in the lattice, it must be ignored. Only the lowest, or minimal, meets should be taken.

In the figure 4.2 object node e with intent $A = \{a, b, c, d\}$ was inserted into a lattice (in the following, the intent attributes of all objects to be inserted are shaded in grey). The set of the meets of all the subsets of A is $\{a, b, c, d, m_1, m_2, m_3\}$. Since a, b, c, d, m_1 are covered by either m_2 or m_3 they can be ignored and e only connected to m_2 and m_3. By inspection, it can be verified that the resulting line diagram is indeed a lattice in that the supremum and infimum of any pair of concepts are unique (keep in mind that the unit and zero nodes were omitted in the figure but are implied).

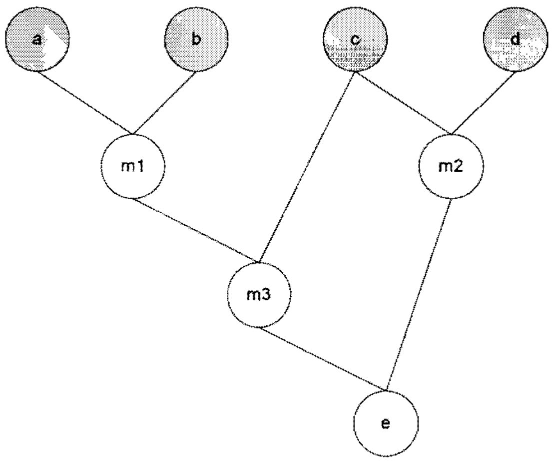

Figure 4.2: node e with intent $A = \{a, b, c, d\}$ was inserted into a lattice by connecting it to m_2 and m_3

This observation suggests a possible lattice construction algorithm. The approach is to find the minimal meets of Intent(o) (i.e. all the meets of all possible subsets of Intent(o), excluding the unit node, not spanned by another meet). This set of nodes can be found by computing the set of meets of all possible subsets of Intent(o) and then removing the zero node and any other node that is spanned by another node lower down in the lattice. Note that this corresponds to the definition of the approximate intent representatives of Intent(o) or AIR(L, Intent(o)) defined in chapter 2.

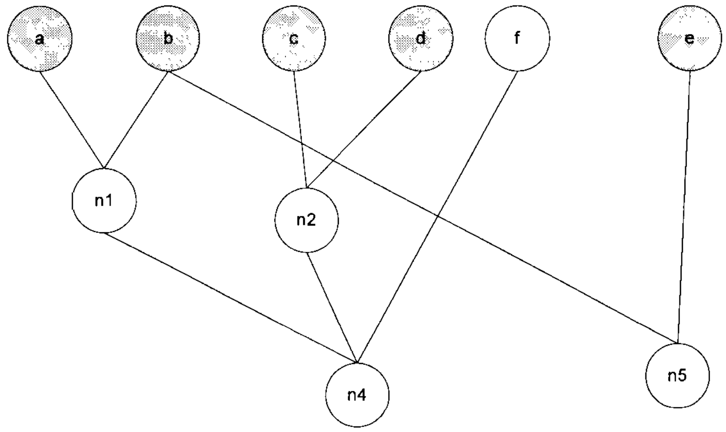

Figure 4.3: Lattice before inserting node m with intent $Intent(m) = \{a, b, c, d, e\}$ to create lattice in figure 4.4

This approach to a lattice construction algorithm does however not always function correctly. Consider the lattice in figure 4.3 and suppose that the node m with intent Intent(m) = {a, b, c, d, e} is inserted into the lattice. Using this approach it creates the lattice in figure 4.4, i.e. because the set of approximate intent representatives of {a, b, c, d, e} in figure 4.3 is $\{n_4, n_5\}$, m is connected to both n_4 and n_5 as shown in figure 4.4. (To aid the readability of the figures, the newly inserted nodes are shown in black.) On closer inspection, we see that m has now gained an extra attribute in its intent namely f via node n_4 (i.e. instead of m's intent being {a, b, c, d, e} as was intended, it is in fact {a, b, c, d, e, f} in figure 4.4). It thus seems as if this approach only works when the intent representative concepts span only attributes in Intent(m) (i.e. when they are exact). If not, then the intent of the new node could unintentionally be extended.

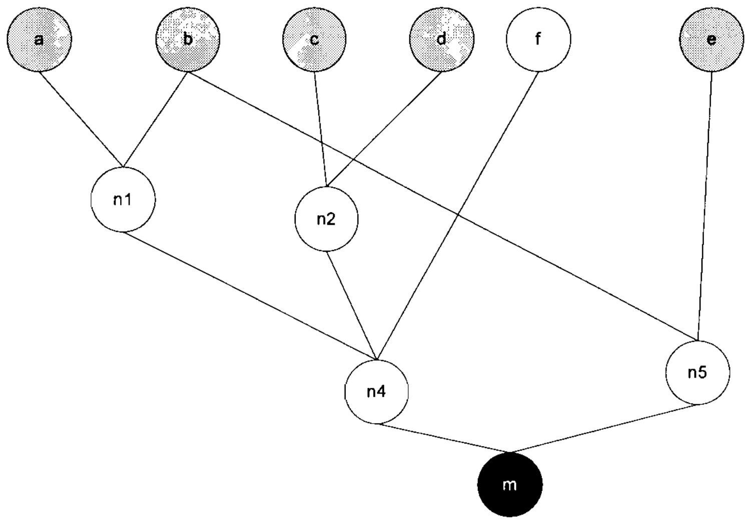

Figure 4.4: Lattice after inserting node m with $Intent(m) = \{a, b, c, d, e\}$, but showing that m now has f in its intent in addition

Since n_4 is not an *exact meet* of Intent(m) in figure 4.3, it cannot be connected directly to the new node. We might be tempted to connect m to n_1, n_2 and n_5, leaving us with the graph in figure 4.5 below. But as indicated using thick arcs, both m and n_4 are lower bounds of $\{n_1, n_2\}$. Since the greatest lower bound of $\{n_1, n_2\}$ is non-unique, the lattice property does not hold and this approach is therefore also not correct.

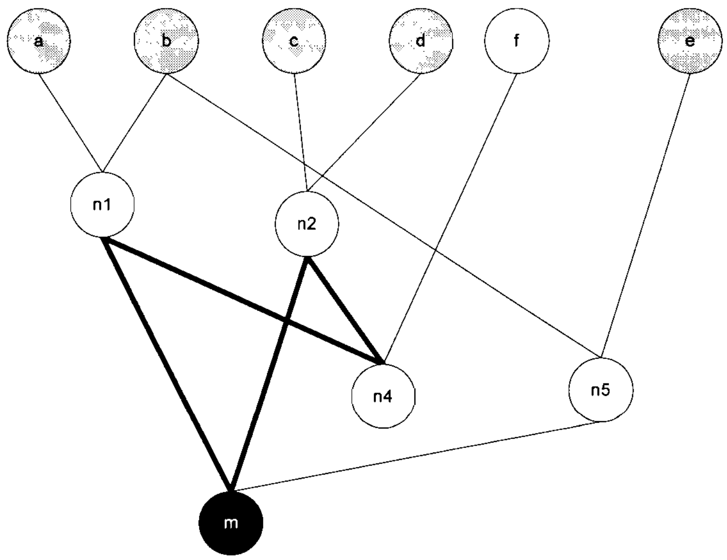

Figure 4.5: Connecting m to n_1 and n_2 creates multiple greater lower bounds of $\{n_1, n_2\}$

The solution to the problem lies in the creation of a new intermediate node n_3 spanning n_1 and n_2 and connecting n_4 and m to n_3 as in figure 4.6. In doing so arcs $\langle n_4, n_1\rangle$ and $\langle n_4, n_2\rangle$ had to be removed and the new arcs $\langle n_3, n_1\rangle$, $\langle n_3, n_2\rangle$, $\langle n_4, n_3\rangle$ and $\langle m, n_3\rangle$ had to be created.

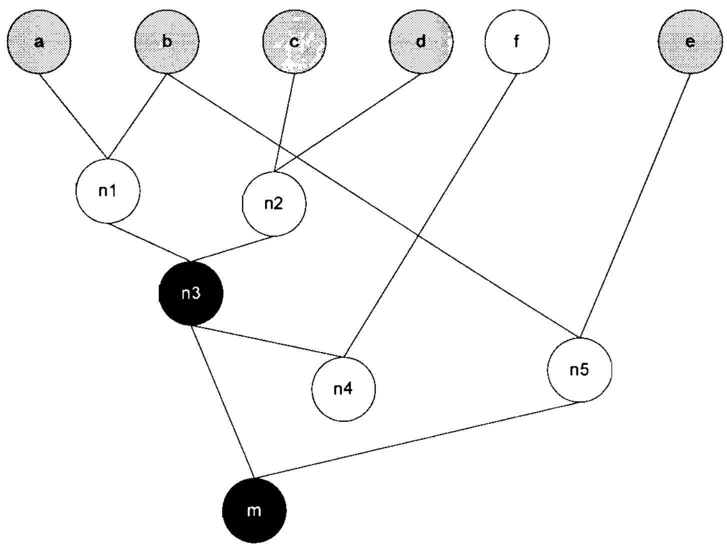

Figure 4.6: To insert m into the lattice a new node n_3 needs to be created

Although we define the AddAtom algorithm in a more formal way in the next section, the key to the algorithm is that new nodes can be directly linked to their exact intent representatives. Additional nodes must be inserted when the intent representatives are approximate. By doing this, we are in effect creating the exact meets of the intent when they do not already exist in the lattice.

The algorithm we are now informally defining needs one extra part: the process of creating the exact meets must be recursively applied. This is demonstrated in the following example where a new node m with intent {a, b, d, e, f, h, g} must be inserted into the lattice in figure 4.7.

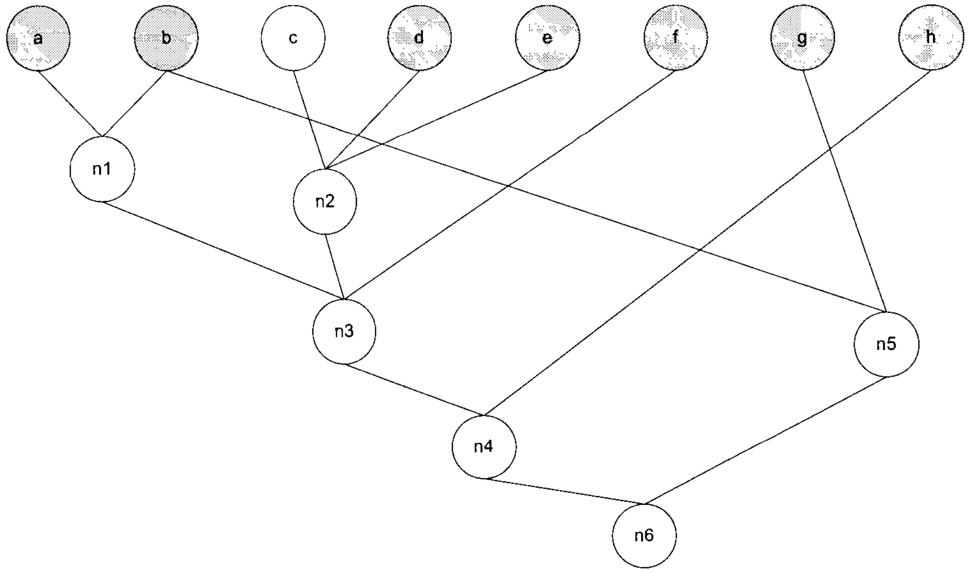

Figure 4.7: A lattice where a new node m with intent {a, b, d, e, f, h, g} must be inserted

The meet of {a, b, d, e, f, h, g} in the lattice in figure 4.7 is $\{n_6\}$. Since n_6 is approximate (it spans c in addition to {a, b, d, e, f, h, g}), a new node (n_{10}) with intent {a, b, d, e, f, g, h} must be created above n_6. This node creates an exact meet to which m can be connect to. However the same reasoning needs to be applied to n_{10} the insertion of itself – it should also be connected the minimal meets of {a, b, d, e, f, g, h} and these meets should be exact. However, when calculating the minimal meets of {a, b, d, e, f, g, h}, n_6 and nodes below it needs to be excluded from consideration. The set of minimal meets of {a, b, d, e, f, g, h} excluding n_6 is therefore $\{n_4, n_5\}$. Node n_5 is an exact meet of {a, b, d, e, f, g, h} and n_{10} can be directly connected to it. Node n_4 is however not exact and an additional node needs to be created above n_4 in the same way n_{10} was created above n_6 (refer to figure 4.8). The insertion of n_{10} can also be viewed as the insertion of an object with the intent of {a, b, d, e, f, h, g} into the sublattice of which n_6 is the zero node. In this context (i.e. n_6 is considered to be the zero node of a sublattice) the set of approximate intent representatives of {a, b, d, e, f, h, g} is $\{n_4, n_5\}$.

Recursively continuing with this process we see that n_3 and n_2 are also approximate meets. Each time such approximate meets are encountered a node is created above the approximate meet. The intent of the new node is that subset of the intent of the approximate meet where only those attributes that are in the intent of the original object (m) are kept. The nodes n_9, n_8 and n_7 are therefore created above the approximate meets n_4, n_3 and n_2 respectively resulting in the lattice in figure 4.8.

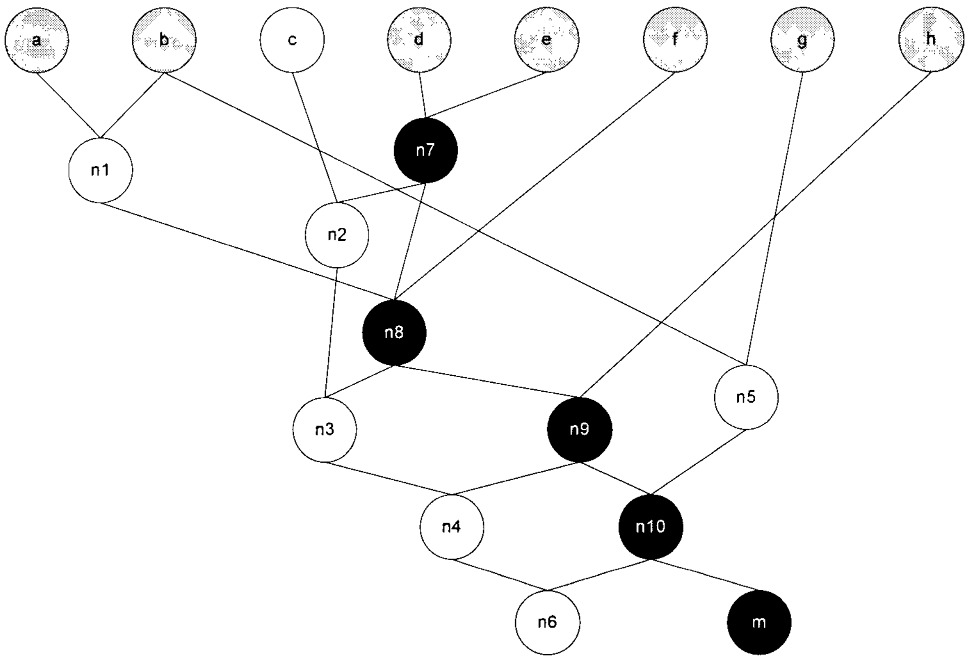

Figure 4.8: The lattice of figure 4.7 after inserting node m with intent $\{a, b, d, e, f, h, g\}$

4.2 INTENT- AND EXTENT REPRESENTATIVE OPERATIONS AND LATTICE CONSTRUCTION

At a high level of abstraction, lattice construction algorithms may be thought of as searching the space of all concepts (i.e. $\mathcal{P}(O) \times \mathcal{P}(A)$) to find all formal or EA-formal concepts. This can for example be done by intersecting the intents of the concepts and searching for sets of attributes each of which are not already present as the intent of some other concept. At a somewhat lower level of abstraction, an incremental lattice construction algorithm that inserts a new object o into a lattice L_i to create a new lattice L_{i+1} may be described (Valtchev and Missaoui 2001) as a search for three sets of concepts in L_i: *generator* concepts, $G(o)$, that give rise to new concepts; modified concepts, $M(o)$, whose arcs must be modified in order to integrate o into their extents; and old concepts, $U(o)$, that remain entirely unchanged. In addition, a set of new concepts $N(o)$ to be inserted into L_i to give L_{i+1} must also be constructed.

The discussion below will indicate that the intent representative operations may be deployed to identify generator, modified, old concepts and new concepts, and may consequently be used to construct concept lattices.

The intent representative operations reflect some of the properties of a lattice and its line diagram. For any concept c in a lattice L (potentially a concept sublattice), $\mathrm{EIR}(L, \mathrm{Intent}(c), c)$ **is** the set of parent concepts of c and therefore defines the cover relationships of c. This property is due to EIR being the minimal meets, not spanning c, that span only contains subsets of $\mathrm{Intent}(c)$ in their intents. Similarly $\mathrm{EER}(L, \mathrm{Extent}(c), c)$ is the set of child concepts of c.

However, this property only holds for concepts that already belong to an existing lattice, L_i. When inserting a *new* object, o, into L_i to create L_{i+1}, it will not necessarily be true that the set $\mathrm{EIR}(L_i, \mathrm{Intent}(o), \mathrm{Inf}'(\mathrm{Intent}(o)))$ represents *all* the parent concepts of o in L_{i+1}. Indeed, an incremental lattice construction algorithm will invariably have to construct (or 'spawn') additional intermediate concepts that are not yet part of L_i. This is in order to achieve the objective that $\mathrm{EIR}(L_{i+1}, \mathrm{Intent}(o), o)$ is the set of parent concepts of c in L_{i+1}.

Furthermore, these additional intermediate concepts and their associated cover relationships in the new structure also have to comply with the lattice property in that any pair of concepts must have a unique infimum and supremum. Therefore in addition to creating the parent concepts of o, other concepts could be created recursively and connected higher up in the lattice in order for this uniqueness property to hold.

It can be shown that an incremental lattice construction algorithm that inserts an object o into a lattice L_i to give L_{i+1}, merely needs to intersect the intent of o with the intent of current concepts in L_i to determine the intent of concepts of L_{i+1}. Any intents of derived in this way that are not the intents of concepts in L_i are that of new concepts that must be added to L_i to derive L_{i+1}. Put differently, the intent of each of these new concepts corresponds to the intersection of Intent(o) with one of the concepts in G(o), the generator concepts for o. In fact, this property is precisely what determines a generator concept for o – that its intersection of its intent with Intent(o) gives the intent of a new concept. This is however a computationally inefficient way to construct lattices and hence the search for efficient construction algorithms.

For simplicity, we will not consider contexts and their corresponding lattices in which the intent of an object is a subset of the intent of some other object (i.e. it is assumed that objects are not comparable). Also assume that the extent of an attribute is not a subset of the extent of any other attribute (i.e. it is assumed that attributes are not comparable). In other words only contexts where the attributes and objects are the co-atoms and atoms respectively of the FCA lattice, and where the FCA lattice is therefore isomorphic to the EA-lattice are considered. This will not detract from validity of the discussion but will prevent the discussion from being cluttered by having to consider some exceptions associated with such contexts.

Consider inserting an object o into L_i. The trivial case is when there are no generator concepts except for the zero concept, 0_L. In this case all concepts in AIR(L_i, Intent(o), 0_L) are exact meets. The object should be inserted, as an atom, above 0_L and connected to its parent concepts as given by EIR(L_i, Intent(o), 0_L). 0_L is the object's only child concept. (Note that this is by virtue of the simplification of the context as described in the previous paragraph.) The extent of each concept in M(o) also needs to be updated as a result of the insertion of o.

If EIR(L_i, Intent(o), 0_L) $\neq$ AIR(L_i, Intent(o), 0_L) then there is at least one concept in L_i that is the meet of a subset of Intent(o) that spans attributes other than those in Intent(o) (i.e. the meet is not exact). All non-exact meets are elements of the set T in the definition of EIR(L_i, Intent(o), o) (refer to section 2.9). For each such non-exact meet, a new concept must be created whose intent corresponds to the intent of the generator concept less the additional attributes. Each such meet is a concept in G(o). Therefore, if EIR(L_i, Intent(o), o) $\neq$ AIR(L_i, Intent(o), o), generator concepts of o do exist in L. Indeed the concepts in the set AIR(L_i, Intent(o), o) - EIR(L_i, Intent(o), o) are all generator concepts, the intersection of the intent of each of these generator concepts with Intent(o) does not represent the intent of any concept already contained in L. (If it did, then that concept would be an element of EIR or AIR.) (Note that these are not the *only* generator concepts as explained below.) An incremental concept lattice construction algorithm can thus compute the minimal (but not all) concepts in G(o) if it can compute the intent representative operations. For the purposes of this discussion, it will be assumed that efficient algorithms to calculate AIR and EIR are indeed available.

The next 'level' of the elements of G(o) can be found by using the intents of the minimal concepts in G(o) restricted to Intent(o) as a generating set and then calculating their respective EIR and AIR sets (i.e. using Intent(g) $\cap$ Intent(o), g $\in$ G(o) for calculating EIR and AIR). This strategy can be recursively applied to calculate all elements of G(o).

If all concepts in $G(o)$ are known, then the new concepts to be inserted can be determined as follows. Each element g of $G(o)$ gives rise to a new concept $n \in N(o)$ ($N(o)$ being the set of new concepts inserted in L_i to yield L_{i+1}) with $Intent(n) = Intent(g) \cap Intent(o)$ and $Extent(n) = Extent(g) \cup \{o\}$. Some of the parent concepts of n could be newly created concepts of $N(o)$ in L_{i+1} higher up in the lattice whilst the others are elements of $EIR(L_i, Intent(o), g)$. Before connecting n, all elements of $N(o)$ must be generated since n might be connected to one of them. The child concepts of n are given by $EER(L_{i+1}, Extent(n), n)$. g will be one of the child concepts but it could have additional child concepts. Each of these child concepts will be in $N(o)$ corresponding to another generator concept lower down in the lattice in a similar way as the parent concepts.

From this description it thus follows that the set of concepts in $G(o)$ is partially ordered. The concepts in $M(o)$ are all the exact meets of subsets of $Intent(o)$ in L_0. Elements of $G(o)$ are all approximate meets of $Intent(o)$. The elements of $U(o)$ are those concepts not in either $G(o)$ or $M(o)$.

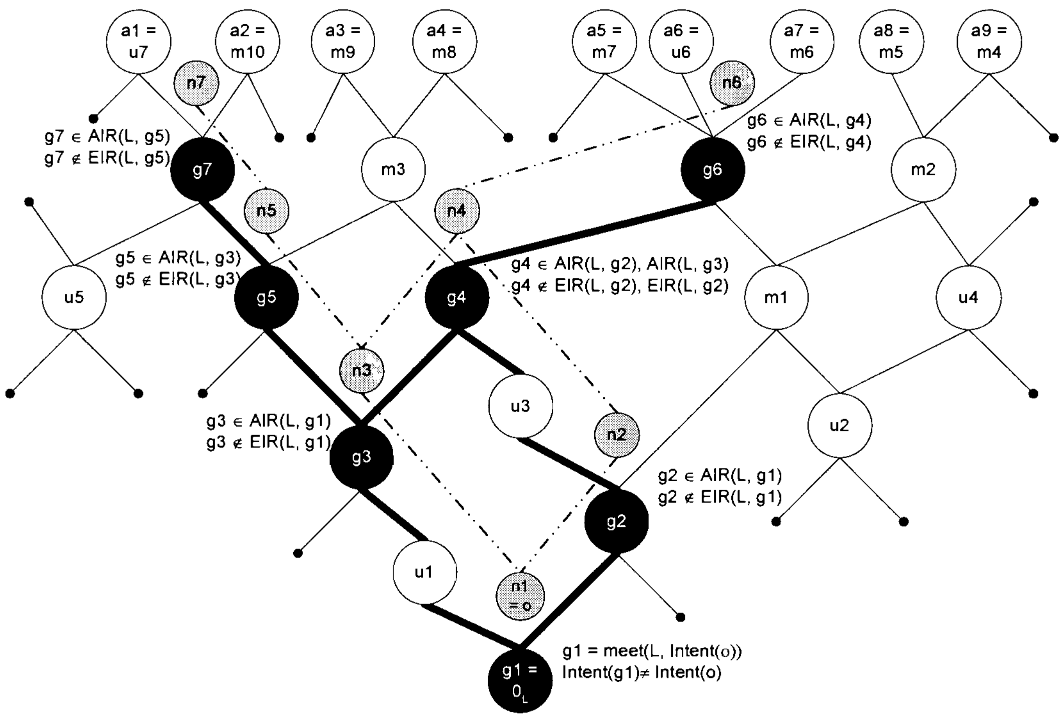

Figure 4.9: The relationship between $G(o)$, $M(o)$, $U(o)$ and $N(o)$ when inserting o with $Intent(o) = \{a_2, a_3, a_4, a_5, a_7, a_8, a_9\}$ into the lattice

Figure 4.9 shows the lattice concepts of L_i as larger circles. They comprise of $U(o)$, $M(o)$ and $G(o)$. Membership of a particular set is indicated by the prefix u, m and g in the concept labels respectively and attributes are prefixed by a. In the example, object o with $Intent(o) = \{a_2, a_3, a_4, a_5, a_7, a_8, a_9\}$ is to be inserted. The elements of $G(o)$ form a partial order, indicated by the thick arcs, with g_1 as zero concept and 1_L as unit concept. The rest of the lattice concepts are not shown and are indicated by thin arcs that do not end/start in concepts. These concepts are members of $U(o)$ and will remain unchanged. The elements of $M(o)$ are all located *above* the largest concepts of $G(o)$. The elements of $N(o)$ are superimposed on the concepts in L_i and shown as smaller, grey shaded, concepts connected by dotted arcs. Each element of $N(o)$ is shown above its respective generator concept. Note that the concepts of $N(o)$ are not yet properly connected into L_i to form L_{i+1}. As explained, $g_1 \in G(o)$ and is in fact 0_L and $n_1 \in N(o)$ is in fact the object o.

These ideas are made more explicit in the formulation of the AddAtom lattice algorithm defined in the next section.

4.3 DEFINITION OF THE ADDATOM ALGORITHM

In this section the algorithm hinted at in the two previous sections is formally defined using pseudo code. For the purpose of reference we call the algorithm AddAtom since it inserts an atom concept (i.e. an object) above the zero concept into the lattice. As defined the algorithm is conceptually simple but very inefficient. Efficient versions of the algorithm are discussed and defined later on in the text. Once again we only consider contexts that have objects that are unrelated to other objects and attributes that are completely unrelated to other attributes as explained earlier. In the corresponding lattice all the objects are thus atoms and the all attributes, coatoms.

The algorithm involves the recursive application of the ideas presented in the previous section. The algorithm is initiated by a set of attributes representing the intent of the object to be inserted (i.e. as a new atom) as well as the zero concept as the first generator concept. Each recursive AddAtom call creates aNewConcept with Intent(aNewConcept) = anAttributeSet. After each recursive call of the algorithm a new concept has been inserted into the lattice above the generator concept. This newly inserted concept has also been properly connected to its parent concepts (possibly involving further recursive AddAtom calls to create the necessary concepts). The called function returns this newly created concept and the calling function inserts this concept into the upper cover of its respective aNewConcept. Thus the recursive calls construct the additional concepts required for the insertion of the object. In this way there is no need to separately compute the covers of the newly inserted and modified concepts since the nature of the intent representative sets as traversed by the recursive calls already indicate these relationships (as depicted in the structure of G(o) in figure 4.9).

Using parameter names to imply types the AddAtom algorithm is defined as follows:

```
//===================================================================
Function AddAtom (L, anAttributeSet, aGeneratorConcept)
      Return aNewConcept
//===================================================================
//Pre-condition:
// L is a partial order such that:
// 1) anAttributeSet is a set of attributes
// 2) UpwardClosure(L, aGeneratorConcept) is a complete sublattice
// 3) Meet(L, anAttributeSet) = aGeneratorConcept
// 4) aGeneratorConcept is a generator concept for
//     anAttributeset and an approximate meet of anAttributeSet
//===================================================================
//Post-condition: L is a minimally updated in such a way
// to ensure that:
// 1) UpwardClosure(L, aGeneratorConcept) remains a sublattice
// 2) Meet(L, anAttributeSet)= aNewConcept (an exact meet, AIR=EIR)
// 3) aNewConcept covers only aGeneratorConcept and nothing else
// 4) All generator concepts above aGeneratorConcept
//    have been visited and the corresponding
//    new concept has been created an appropriately
//    linked into L
//===================================================================
ApproxMeets =
   AIR(L, anAttributeSet, aGeneratorConcept) -
   EIR(L, anAttributeSet, aGeneratorConcept)
// Remove all elements of EIR from AIR
// Pre-condition 2 guarantees that the meets are unique
// Next, generate N(o)
Do While (ApproxMeets ≠ Ø)
   Select and mark any x ∈ ApproxMeets
   SubAttr = anAttributeSet ∩ x.Intent
   bNewConcept = AddAtom(L, SubAttr, x)
   Recompute ApproxMeets
   Remove all marked concepts from ApproxMeets
Od
// Post-condition 4 achieved and AIR = EIR
aNewConcept = CreateConcept(L)
aNewConcept.Extent = aGeneratorConcept
aNewConcept.Intent = anAttributeSet
// Next, connect elements of N(o) to aNewConcept
For ∀ x ∈ EIR(L, anAttributeSet, aGeneratorConcept)
   CreateArc(L, aNewConcept, x)
   //Assume no effect if arc already exists
   DeleteArc(L, aGeneratorConcept, x)
   //Assume no effect if arc does not exist
Rof
// Next update the extents of N(o) and M(o)
If aGeneratorConcept = 0_L then
   For ∀ x ∈ UpwardClosure(L, aNewConcept)
      x.Extent = x.Extent ∪ {aNewConcept}
   Rof
Fi
// Post-condition 1 & 2 achieved
CreateArc(L, aGeneratorConcept, aNewConcept)
// Post-condition 3 & 5 achieved
Return aNewConcept
End AddAtom
//===================================================================
```

Thus, to incrementally insert a new object o into a lattice for the context $\langle A, O, I\rangle$ the function call $\mathrm{AddAtom}(L, \mathrm{Intent}(o), 0_L)$ would be used. Note that L is passed as an in/out parameter. It is assumed that the individual attributes of the object o are already present in the lattice (i.e. as coatoms).

As indicated a list of marked concepts needs to be kept in order that such concepts are not revisited in the Do While…Od loop.

To operate on arbitrary contexts the AddAtom algorithm should be slightly extended to consider the following cases:

- The object is the first to be inserted into an empty L_0.
- The object to be inserted into the lattice is in fact not an atom in L_i (i.e. $\mathrm{Intent}(o)$ is a subset of some other object's intent).
- The object has same intent as another object in the context.
- The attributes of the object are not all coatoms.
- More than one attribute may correspond to a single concept in L_i (i.e. the extent of two attributes is the same).
- Some of the attributes in $\mathrm{Intent}(o)$ do not already exist in L_i.
- Some modifications are required for FCA lattices (since all objects are not atoms and all attributes not coatoms).

In addition to these the AddAtom algorithm can be modified to operate on compressed pseudo-lattices (refer to chapter 6) in that it respects the virtual arcs and compressed pseudo-lattice properties and does not assume the existence of all formal concepts in L_i.

4.4 ADDATOM EXAMPLE

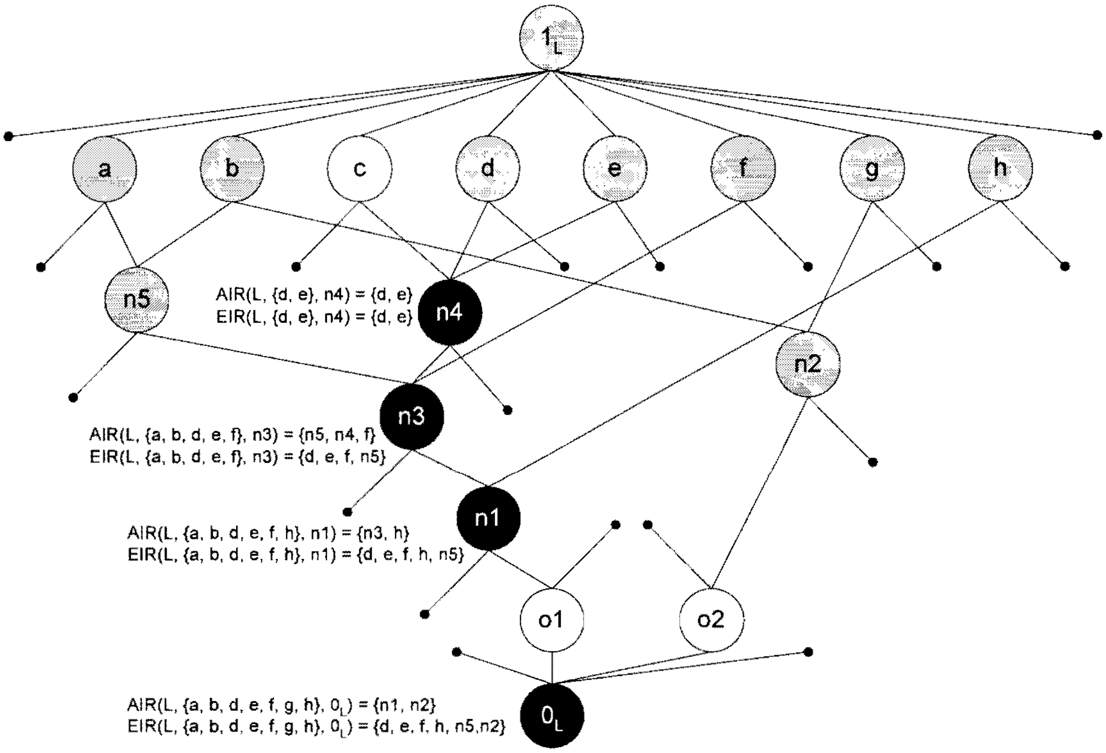

Figure 4.10: A lattice before inserting o_3 with $Intent(o_3) = \{a, b, d, e, f, g, h\}$ indicating $G(o)$ as well as the AIR and EIR sets of elements of $G(o)$

As an example consider inserting object o_3 with $\text{Intent}(o_3) = \{a, b, d, e, f, g, h)$ into the lattice, L, in figure 4.10. Since the algorithm does not consider and visit irrelevant concepts, only the relevant part of L is shown – the relationships to the rest of L are shown by arcs that do not terminate in concepts.

L is an in/out parameter in the algorithm. Thus, throughout the algorithm, operations use L as it exists at that point in the computation - not in its state when that given level of recursion was invoked with L as a parameter.

The algorithm begins with the function call $\text{AddAtom}(L, \{a, b, d, e, f, g, h\}, 0_L)$. ApproxMeets has to be computed, and this requires that both $\text{AIR}(L, \{a, b, d, e, f, g, h\}, 0_L)$ and $\text{EIR}(L, \{a, b, d, e, f, g, h\}, 0_L)$ have to be computed. In this case $S = \{1_L, a, b, d, e, f, g, h, n_5, n_4, n_3, n_2, n_1\}$ (refer to section 2.9 for the definition of AIR and EIR). These concepts are shown in black or grey in figure 4.10. Other concepts are in white. The concepts in black are generator concepts as will become clear later. n_1 and n_2 are the two minimal concepts in S, therefore $\text{AIR}(L, \{a, b, d, e, f, g, h\}, 0_L) = \{n_1, n_2\}$.

In order to find $\text{EIR}(L, \{a, b, d, e, f, g, h\}, 0_L)$, we see that $T = \{n_4, n_3, n_1\}$ and therefore $S - T = \{1_L, a, b, d, e, f, g, h, n_5, n_2\}$. Thus, $\text{EIR}(L, \{a, b, d, e, f, g, h\}, 0_L)$, (the set of minimal concepts, excluding 0_L, in $S - T$) is $\{d, e, f, h, n_5, n_2\}$. As a result ApproxMeets (AIR – EIR) is $\{n_1\}$. n_1 is therefore a generator concept.

The first loop of the algorithm is thus executed, where $x = n_1$. $\text{Intent}(L, n_1) = \{a, b, c, d, e, f, h\}$ and $\text{SubAttr} = \{a, b, d, e, f, h\}$. Thus, $\text{AddAtom}(L, \{a, b, d, e, f, h\}, n_1)$ is recursively called. Note that n_1 is an approximate meet of $\{a, b, d, e, f, g, h\}$ since it also spans the attribute c. To create an exact meet that does not span the additional attribute, c. The algorithm searches for any additional approximate meets above n_1 and creates additional concepts that will form exact meets.

In tracing the function call AddAtom(L, {a, b, d, e, f, h}, n_1) we see that AIR(L, {a, b, d, e, f, h}, n_1) = {n_3, h} and EIR(L, {a, b, d, e, f, g, h}, n_1) = {d, e, f, h, n_5, n_2} so that ApproxMeets = {n_3}. SubAttr = {a, b, d, e, f} with n_3 being an approximate meet of SubAttr, again spanning c in addition. n_3 is thus a generator concept. AddAtom(L, {a, b, d, e, f}, n_3) is therefore recursively called to create an exact meet above n_3.

AddAtom(L, {a, b, d, e, f}, n_3) calculates AIR(L, {a, b, d, e, f}, n_3) = {f, n_5, n_4} and EIR(L, {a, b, d, e, f}, n_3) = {d, e, f, n_5}, so that ApproxMeets = {n_4}.

Once again n_4 is a generator node and AddAtom(L, {d, e}, n_4) is called recursively. Since AIR(L, {d, e}, n_4) = EIR(L, {d, e}, n_4) = {d, e}, ApproxMeets = ∅ and the algorithm progress past the while loop to create n_6 whose intent is to become {d, e} (figure 4.11). Moving to the next loop of AddAtom EIR(L, {d, e}, n_4) = {d, e} and therefore arcs are created between n_6 and d and n_6 and e. n_4 disconnected from both d and e. Finally after completion of the for loop an arc is created between n_4 and n_6 and AddAtom(L, {d, e}, n_4) terminates with n_6 as the result which is passed back to AddAtom(L, {a, b, d, e, f}, n_3).

AddAtom(L, {a, b, d, e, f}, n_3) now creates n_7 and calculates EIR(L, {a, b, d, e, f}, n_3) = {n_6, n_5, f} (n_6 being the newly created exact meet). It then creates arcs from n_7 to n_5, n_6 and f. The arcs from n_3 to n_5 and f are deleted. An arc is created between n_3 and n_7 and the function returns n_7 as the result.

AddAtom(L, {a, b, d, e, f, g, h}, n_1) creates n_8 and since EIR(L, {a, b, d, e, f, g, h}, n_1) = {h, n_7} arcs from each to n_8 are created. The arc between n_1 and h is deleted. An arc between n_1 and n_8 is created and AddAtom(L, {a, b, d, e, f, g, h}, n_1) terminates with n_8 as result.

Finally AddAtom(L, {a, b, d, e, f, g, h}, 0_L) creates o_3 and create arcs between o_3 and n_2 and n_8. Since o_3 is a newly inserted object it is added to the extent of all the concepts above it. 0_L is connected to o_3. This concludes the recursive AddAtom calls and AddAtom returns the inserted object o_3 to the calling function. Since L was an in/out parameter, it now refers to the newly created lattice.

The resulting EA-lattice is shown in figure 4.11 with the newly created concepts shown in grey and their corresponding generator concepts in black. The AddAtom function calls are also shown next to the respective generator concepts.

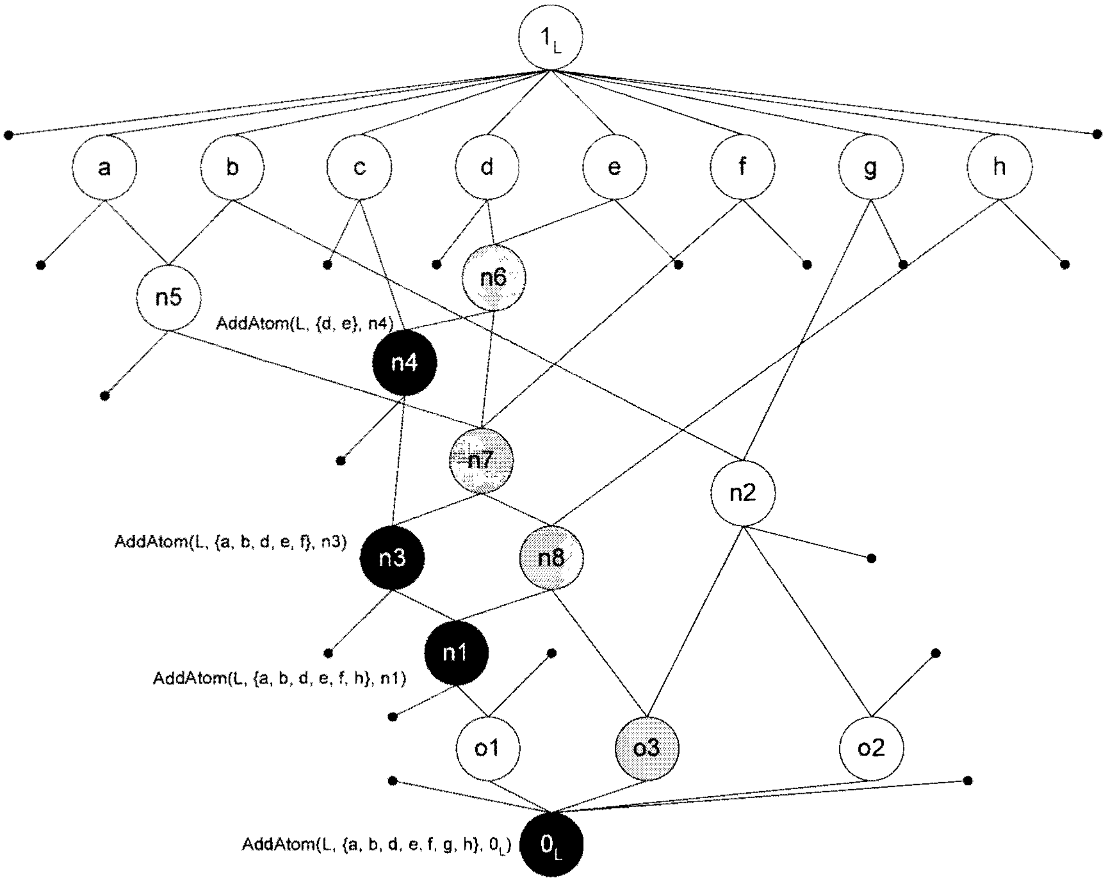

Figure 4.11: The AddAtom example after inserting o_3 with $Intent(o_3) = \{a, b, d, e, f, g, h\}$, $G(o)$ and $N(o)$ as well as the recursive AddAtom calls are indicated

AddAtom thus starts at the bottom of the lattice at the zero concept and traverses the lattice upward, creating new concepts associated with 'approximate' meets. The new concepts form exact meets of the intent of the object. The recursion terminates when AddAtom encounters only 'exact' meets (i.e. elements of $M(o)$) to which the newly created concepts are connected. In this way the recursive calls efficiently search the lattice for generator concepts and, whilst doing so, use the inherent structure of L_0 to search for, create and connect the concepts of L_1.

The example also shows how the structure and ordering of concepts in L_0 can be used to efficiently eliminate many concepts in the lattice from consideration by using the AIR and EIR operations. Some incremental lattice construction algorithms resort, in a sense, to a more brute force approach in considering a much larger set of concepts in order to test for generation concepts or in order to intersect the intent of the object with these concepts.

4.5 AN ALGORITHM FOR AIR AND EIR

It might be argued that the AddAtom algorithm is merely a restatement of an incremental lattice construction algorithm in terms of AIR and EIR but that the calculation of AIR and EIR is computationally inefficient. This research indicates that there are indeed efficient algorithms for calculating AIR and EIR but these rely on the explicit representation of the line diagram or cover relationship as a data structure.

One way of efficiently calculating AIR and EIR is to use the concept of marker propagation in which so-called "markers" are propagated downward along *all* paths leading from each of the attributes of the object o. Afterwards the number of markers that have accumulated on each of the concepts is counted. The number of markers thus indicates how many attributes of o a concept has in its intent. Concepts with zero markers therefore need not be considered as candidates for being minimal meets in AIR or EIR. Concepts with a higher number of markers are lower down in the lattice than those with a lower number of markers. Furthermore, there will be many concepts that have the same number of markers. The number of markers increases as one moves down in the lattice.

There are three key observations to finding AIR (and EIR) using marker propagation. The first key observation is that any concept that has somewhere below it in the lattice another concept with more markers than itself is not a candidate for AIR, since it can not be minimal. The second key observation is that a concept is only a candidate if it does not have a parent concept which has the same number of markers as itself (i.e. if it is the highest concept with that number of markers and has no parent with the same number of markers). This is because if any concept has a parent concept above it with the same number of markers, it cannot be a greatest (i.e. highest) lower bound of a subset of Intent(o). The third observation is that when searching for candidate concepts by starting with those with the highest number of markers and eliminating all concepts above and below them from consideration, all candidate concepts will be found.

Using markers one thus has to search for all concepts that have the largest number of markers accumulated upon them but that have no concept below them with more markers. All such concepts are candidate concepts, but only those that have no concept above them with the same number of markers are elements of AIR.

Figure 4.12 is part of a lattice before inserting object o into it. Suppose Q is the set of attributes associated with o and markers are propagated down from each attribute. The concepts are labelled by the number of markers accumulated on them (i.e. the number of attributes of Q it spans). Arcs to the rest of the lattice are shown as lines ending in small circles without concept numbers. Those arcs ending in filled/solid circles indicate arcs to attributes in Q and those to unfilled circles indicate arcs to unique attributes not in Q. The marker count is therefore the number of filled small circles above each concept.

To search for AIR(Q) the set of concepts with the highest number of markers (5 markers) is considered. In this case the set is $\{n_{21}, n_{24}, n_{25}, n_{26}\}$. n_{25} and n_{26} have a concept above them with the same number of markers so they can be discarded from the set, leaving $\{n_{21}, n_{24}\}$. Next we eliminate all the concepts in n_{21} and n_{24}'s upward and downward closure from consideration and continue searching for concepts with the highest number of markers. In the remaining concepts, n_{27} has the highest number of markers with 4. After eliminating its upward and downward closures from consideration the only concepts with more than zero markers that remain are n_{12}, n_{17}, n_{18} and n_{23} with three markers each. Since n_{17}, n_{18} and n_{23} have concepts above them with the same number of markers, n_{12} is the last remaining element of AIR(Q). Therefore AIR(Q) = $\{n_{12}, n_{21}, n_{24}, n_{27}\}$. These concepts are shown in black. They are all generator concepts of o but are not the only generator concepts of o (the other generator concepts are n_4, n_6, n_7, n_8, n_{10}, n_{14} and n_{19}).

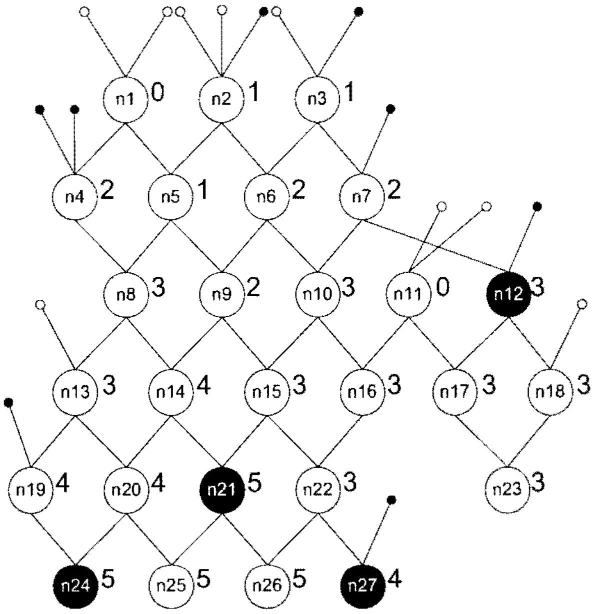

Figure 4.12: Part of a lattice before inserting object o into it showing $AIR(o)$ in black. Each concept (n_1 to n_{27}) is labeled with the number of markers / attributes of o that has accumulated on it.

This process formalised in the following algorithm:

```
//=================================================================
Function AIR(L, anAttributeSet) Return aConceptSet
//=================================================================
//Pre-condition:
// L is a concept lattice with anAttributeSet a non-empty subset of
// L's attributes
//=================================================================
//Post-condition:
// aConceptSet contains the minimal (possibly approximate) meets
// of anAttributeSet or AIR(L, anAttributeSet)
//=================================================================
NotVisited = ∅
MaxAttr = 0
Let attrCount[c] = 0 for all c ∈ L
For ∀ a ∈ anAttributeSet
   For ∀ b ∈ DownwardClosure(L, a)
      attrCount[b] = attrCount[b] + 1
      NotVisited = NotVisited ∪ {b}
      If attrCount [b] > MaxAttr then
         MaxAttr = attrCount[b]
      Fi
   Rof
Rof
Candidates = ∅
Do While (NotVisited ≠ ∅ and MaxAttr > 0)
   Let d be any c ∈ NotVisited with attrCount[d] = MaxAttr
   If such a c does not exist then
      MaxAttr = MaxAttr - 1
   Else
      // If d has concepts above it with the same number of markers
      // find the one that is the greatest
      Found = False
      Do While Not Found
         Found = True
         For ∀ p ∈ Parents(d)
            If attrCount[p] = attrCount[d] then
               d = p
               Found = False
               Exit For
            Fi
         Rof
      Od
      Candidates = Candidates ∪ {d}
      // Remove the upward closure of d from further
      // consideration - its elements can not be minimal meets
      UCD = UpwardClosure(L, d)
      NotVisited = NotVisited - UCD
      // Remove any candidates that are greater
      // than d - they can not be minimal
      Candidates = Candidates - UCD
      // Remove all concepts below d since they have MaxAttr
      // markers or have been considered
      DCD = DownwardClosure(L, d)
      NotVisited = NotVisited - DCD
   Fi
Od
Return Candidates
End AIR
//=================================================================
```

The calculation of EIR can be done in a similar way but only concepts that are exact must be considered as candidates. This process can be fast-tracked by eliminating the union of the downward closure of all attributes not in Intent(o) from consideration before propagating the markers. The calculation of AER and EER can be done using the same strategy, but this time propagating markers in the opposite direction and appropriately changing the direction of the relevant operators in the algorithm.

The algorithms of the intent- and extent operations were defined in terms of the closure and set operations. When representing sets as strings of bits in memory, these operations can be very efficiently performed on modern architectures using 32 or 64 bit words. The calculation of AIR and EIR is therefore very efficient.

Since the intents and extents of the concepts in the lattice can be derived from the upward- and downward closures of the concepts in the line diagram, these need not be calculated explicitly.

It is also possible to have attrCount pre-computed when the AIR etc. will be computed for a subset of A. This optimisation is considered in the efficient AddAtom algorithm defined in the next section.

4.6 EFFICIENT ADDATOM ALGORITHM

The AddAtom algorithm as described in section 4.3 is not optimal in terms of efficiency. A number of basic performance improvements can be made on the algorithm. Examples include the possible avoidance of recalculation of ApproxMeet and the processing of the generator concepts in the order of the size of their intent. The calculation of both the exact and approximate intent representative sets can also be computationally inefficient and may duplicate many operations due to the similarity between the two sets. The following algorithm is an efficient version of the AddAtom algorithm of section 4.3. It builds on the ideas of the calculation of AIR and avoids the repeated and calculations of AIR and EIR.

```
//=====================================================================
Function OptimisedAddAtom(aContext) Return aLattice
//=====================================================================
L = CreateEmptyLattice()
1L = NewConcept(L)
0L = NewConcept(L)
0L.Intent = aContext.Attr
For ∀ a ∈ aContext.Attr
   anAttributeConcept = NewConcept(L)
   anAttributeConcept.Intent = {a}
   CreateArc(L, 0L, anAttributeConcept)
   CreateArc(L, anAttributeConcept, 1L)
Rof
For ∀ o ∈ aContext.Obj
   // Calculate attrCount[x], the number of attributes in o.Intent
   // that occur in x.Intent
   Let attrCount[x] = 0 for all x ∈ L
   For ∀ x ∈ L
      attrCount[x] = ||x.Intent ∩ o.Intent||
   Rof
   NewObject = AddAtom(L, o.Intent, 0L, attrCount)
   For ∀ x ∈ UpdwardClosure(NewObject)
      x.Extent = x.Extent ∪ {o}
   Rof
Rof
Return L
End OptimisedAddAtom

//=====================================================================
Function GetMeet(L, target, aConcept, attrCount)
                                   Return returnConcept
//=====================================================================
//Pre-condition:
// L is a concept lattice, attrCount[aConcept] = target
//=====================================================================
//Post-condition:
// returnConcept is the greatest upper bound/concept in L with
// attrCount[returnConcept] = target
//=====================================================================
returnConcept = aConcept
ParentIsMeet = True
Do While ParentIsMeet
   ParentIsMeet = False
   For ∀ Parent ∈ ConceptParents(L, aConcept)
      If attrCount[Parent] = target then
         returnConcept = Parent
         ParentIsMeet = True
         Exit For
      Fi
   Rof
Od
Return returnConcept
End GetMeet

//=====================================================================
Function AddAtom(L, anIntent, GeneratorConcept, attrCount)
                          Return aConcept
//=====================================================================
//Pre-condition:
// 1) UpwardClosure(L, GeneratorConcept) is a complete sublattice
```

```
// 2) GeneratorConcept is the meet of anIntent and is approximate
// 3) attrCount[c] = Intent(c)∩Intent(newObject)
//==================================================================
//Post-condition:
// returnConcept is the greatest upper bound/concept in L with
// attrCount[returnConcept] = target
//==================================================================
CandidateParents = ConceptParents(L, GeneratorConcept)
NewConceptParents = ∅
For ∀ Candidate ∈ CandidateParents
   newIntent = Candidate.Intent ∩ anIntent
   If newIntent ≠ ∅
      If Candidate.Intent ≠ newIntent then
         aMeet = GetMeet(L, ||newIntent||, Candidate, attrCount)
         If aMeet.Intent ≠ newIntent
            // If aMeet is approximate it is a generator concept and an
            // exact meet needs to be created
            aMeet = AddAtom(L, newIntent, aMeet, attrCount)
         Fi
      Else
         aMeet = Candidate
      Fi
      addMeet = True
      For ∀ g ∈ NewConceptParents
         If aMeet.Intent ⊆ g.Intent
            addMeet = False
            Exit For
         Else If g.Intent ⊂ aMeet.Intent then
            NewConceptParents = NewConceptParents - {g}
         Fi
      Rof
      If addMeet then
         NewConceptParents = NewConceptParents ∪ {aMeet}
      Fi
   Fi
Rof
NewConcept = CreateNewConcept(L)
NewConcept.Extent = GeneratorConcept.Extent
NewConcept.Intent = anIntent
attrCount[NewConcept] = attrCount[GeneratorConcept]
For ∀ g ∈ NewConceptParents
   DeleteArc(L, GeneratorConcept, g)
   CreateArc(L, NewConcept, g)
Rof
CreateArc(L, GeneratorConcept, NewConcept)
Return NewConcept
End AddAtom
//==================================================================
```

Some optimisations are still possible, but these do not change the basic structure of the algorithm as stated above. Appendix A contains the pseudo code for one such optimised version of AddAtom that amongst other strategies considers concept parents in descending order of their attrCount value. This allows for the removal of many additional concepts from consideration.

4.7 DISCUSSION

Initially some of the meets of subsets of Intent(o) are approximate meets (i.e. generator concepts). After each completion of a recursive call, additional concepts have been created that would now form the exact meets of those subsets of Intent(o) and replace the approximate meets. The algorithm terminates when all meets of all subsets of Intent(o) are exact with regards to Intent(o). Initially, L is a lattice but as new concepts are generated that are not yet fully integrated to the lattice structure, some parts of L may violate the lattice properties up until the completion of all levels of the recursion. When terminating, the AddAtom algorithm ensures that all concepts in UpwardClosure(L, aGeneratorConcept) form a lattice. Since the first AddAtom call uses 0_L as the generator concept, L will be a lattice when that AddAtom call terminates.

The AddAtom algorithm generates the new concepts and cover relationships in one step and therefore seems to be more focussed than incremental lattice construction algorithms that first generate the concepts and then search and generate the upper covers of concepts using a separate function such as Godin et al. (1991) and Carpineto and Romano (1993, 1996b). Experiments to date (discussed in chapter 5) also suggest AddAtom is more efficient.

The algorithm exploits the relationships between concepts already represented in the lattice to efficiently search for the generator concepts using the intent representative operations. To this extent the algorithm makes explicit use of the line diagram that represents the original lattice structure when searching for G(o) by means of the ordering relationship and the intent representative operations rather than considering all concepts at once in a more brute force search. Indeed, the intent representative operations themselves imply a ordering of the generator and new concepts in L_1.

A very important property of the algorithm is that it can operate on sublattices where the formal concept lattice of a context is not used as input. This is due to the fact that the algorithm is entirely general in not requiring the lattice to have a specific set of atoms or coatoms (i.e. those representing the attributes and objects) but not necessarily that of the formal concept lattice or EA-lattice (similar to those concept sublattices created in compressed pseudo-lattices). Such lattices are not closed with respect to the intersection of intents or the union of extents. The only requirement is that anAttributeSet consists only of coatoms (and not necessarily attribute concepts of the context). The operations used are therefore based on closure operations rather than intersections of intents. Such lattices are for example under consideration in compressed pseudo-lattices where the lattice is not closed with regards to the intersection of intents. Not all lattice construction algorithms are suitable for applications using sub-lattices in this kind of way.

An optimised, object-oriented version of the algorithm was implemented and tested in C++ (chapter 7). In addition, the implementation also implements the concept of a compressed pseudo-lattice (chapter 6). The algorithm therefore takes the existence of virtual- and lattice arcs into consideration during its operation.

Since the direction of the operations can be reversed (e.g. meet replaced by join, EIR by EER, atom by coatom, etc.) a dual for the AddAtom algorithm namely AddCoatom can be defined. In the implementation this was achieved by adding an additional parameter named aDirection to all lattice operations to indicate the direction in which the operation should operate.

The next chapter (chapter 5) analyses the algorithmic performance of the algorithm by comparing the performance of AddAtom to that of other lattice construction algorithms both theoretically and experimentally.