SRI International

Localized Search

Technical Note 476

November 30, 1989

By: Lode Missiaen
Artificial Intelligence Center
Computer and Information Sciences Division
and
National Fund for Scientific Research (Belgium)

**APPROVED FOR PUBLIC RELEASE:
DISTRIBUTION UNLIMITED**

This research has been made possible in part by the National Science Foundation under Grant IRI-8715972 and the National Aeronautics and Space Administration under Contract NCC2-494. The views and conclusions contained in this paper are those of the author and should not be interpreted as representative of the official policies, either expressed or implied, of NSF, NASA, or the United States Government.

SRI International®

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

Abstract

In this report, we describe the search algorithm of the GEMPLAN multiagent planning system. The search algorithm is based upon the GEMPLAN domain description and its localized constraint representation. The problem domain is structured into regions of activity, and each region has its own set of local constraints. The search is a constraint-satisfaction process; it tries to find a plan in each region by satisfying the region's constraints. Therefore, the search space is subdivided into regional search trees. Unfortunately, these search trees cannot be searched independently. However, the situation is much better than global search because GEMPLAN's constraint localization, together with the domain structure, precisely define when the search in one region can affect another region, and hence how control must shift from one search tree to another.

To avoid any confusion, this report does not describe GEMPLAN, but only its generic localized search algorithm. We only explain and abstract features of GEMPLAN on which this search algorithm is based. As a result, this algorithm is applicable to any other constraint-satisfication problem with characteristics similar to GEMPLAN.

Contents

1 Introduction

This report describes the search algorithm of the multiagent planning system GEMPLAN [1] [2] [3] [4]. The search algorithm is based upon the GEMPLAN domain description and its use of localized constraints. A GEM problem domain is structured into regions of activity with each region having its own set of local constraints. Basically, the search is a constraint-satisfaction process that tries to find a plan for each region by satisfying the region's constraints.

In Section 2 we extract the concepts of a GEM domain description that are important for the search and introduce our terminology. Section 3 describes the regional search trees and the flow of control among these trees. Section 4 discusses the implications of regional overlap, and how to deal with it. In Section 5, we study the details of shifting control from a region to its subregions. Section 6 gives a detailed account of the implementation of the search algorithm in Quintus Prolog. Finally, Section 7 reports on current and future extensions to the search algorithm.

To avoid any confusion, this report does not describe GEMPLAN, but only its generic localized search algorithm. We only explain and abstract those features of GEMPLAN upon which this search algorithm is based. As a result, this algorithm is applicable to any other constraint satisfication problem with characteristics similar to GEMPLAN.

2 Domain description

This section explains the structuring of a problem into regions, and introduces some of the terminology that will be used throughout the rest of the report. In the first subsection, we describe these aspects, and in the second subsection, we illustrate them with an example.

2.1 Terminology

A GEM domain is described as a collection of *regions.*[1] Regions are structured by the relation partof(Region,SubRegion), which states that SubRegion is a direct component of Region, called a *subregion.* Partof/2 must define a partial ordering among the regions so that no circularities occur if we construct the transitive closure of partof/2. Let us call this transitive closure t_partof/2. If t_partof(HRegion,LRegion) holds, then LRegion is a *lower-level region* of HRegion, and conversely, HRegion is a *higher-level region* of LRegion. There exists one region with a level higher than any other region: this region is called the *global region.*

Each region has a set of *constraints* and *fixes* associated with it. In GEMPLAN, these constraints are *GEM-constraints* expressed in first-order modal temporal logic. For each type of GEM-constraint, several alternative methods exist to satisfy a violated constraint of that type; such a method is called a *GEM-fix.*[2] The search algorithm is based on some of the properties of regional GEM-fixes and -constraints, which we will describe later. A full account of the GEM-fixes and -constraints is not needed for the purposes of this report; it is enough to know that a constraint in a given region imposes both temporal relations among events and properties that have to hold at certain times, and that a fix introduces new events to establish properties, orders events to satisfy temporal requirements, and binds variables. A *local plan* of a given region consists of the events that occur in that region, together with the relations among these events, and also the relations among events of different subregions and of the region itself.[3] A local plan of a region is *satisfied* if all the constraints of that region are satisfied by the plan. A *regional plan* of a region consists of its local plan, together with all the regional plans of its subregions, which are called *subplans.* A regional plan is satisfied if its local plan and all its subplans are satisfied. The regional plan of the global region is called the *global plan.*

The search will be based on the following important property of fixes and constraints:

[1]GEM makes a distinction between two types of regions, *elements* and *groups.* An accessibility relation defines how elements and groups can affect each other. The search algorithm itself does not rely on this distinction and treats both elements and groups as regions. However, test for accessibility between these regions occur within GEMPLAN's constraint satisfaction algorithms, which then provide the search algorithm with appropriate control flow information.

[2]We prefix *fix* with *GEM* because there is also a *search fix,* which is related to but quite distinctive from a *GEM-fix.* We use simply *fix* when the meaning is clear from the context. The term *constraint* can be used unambiguously.

[3]A plan also contains variable bindings.

> If a fix changes a local plan of a given region, then only constraints in the given region and constraints in the next-higher-level regions can possibly be influenced directly.

Let us call this property *search constraint localization.*[4]

What does the search consist of ? The search has to find a satisfied global plan. Because of search constraint localization, finding a satisfied global plan cannot solely be achieved by searching the regions independently. However, search constraint localization together with the partof/2 relation define precisely when the search in one region can affect another region, and hence how the search control has to shift from one region to another. In fact, the search constraint localization ensures that subplans are unaffected by fixes at the local plan of a given region, and therefore the search constraint localization hints for a bottom-up construction of a regional plan. As we will see in the following sections, the search will be mainly local, and therefore we call it *localized search.*

2.2 Example

In Figure 1, a problem domain is represented. **A** is the global region, with **B** and **C** as its subregions. **D** is a subregion of **C**, but not of **A**. Clearly, **D** is of a level lower than both **C** and **A**. From the search constraint localization property, we derive that local changes in **D** will possibly affect only the constraints in **D** and **C** directly, but not in **A** and **B**. Similar conclusions can be made for the other regions.

The event e_1 belongs to **A**; the events e_2 and e_3 belong to **B**; e_4 belongs to **C**; and e_5 belongs to **D**. The arrow $\longrightarrow$ represents the temporal precedence relation among events. The relation $e_1 \longrightarrow e_2$ belongs to the local plan of **A** because e_2 belongs to the subregion **B**. Similarly, $e_3 \longrightarrow e_4$ belongs to the local plan of **A**, and $e_5 \longrightarrow e_4$ belongs to the local plan of **C**. In Figure 1, the local plans of the regions are

A: $\{ e_1, e_1 \longrightarrow e_2, e_3 \longrightarrow e_4\}$

B: $\{ e_2, e_3, e_3 \longrightarrow e_2 \}$

[4] ***Search constraint localization*** should not be confused with ***constraint localization*** defined in GEM. The main semantic purpose of constraint localization is to limit the scope of constraints. As a result, constraints can be checked and fixed more efficiently. This naturally led to the notion of ***localized search.***

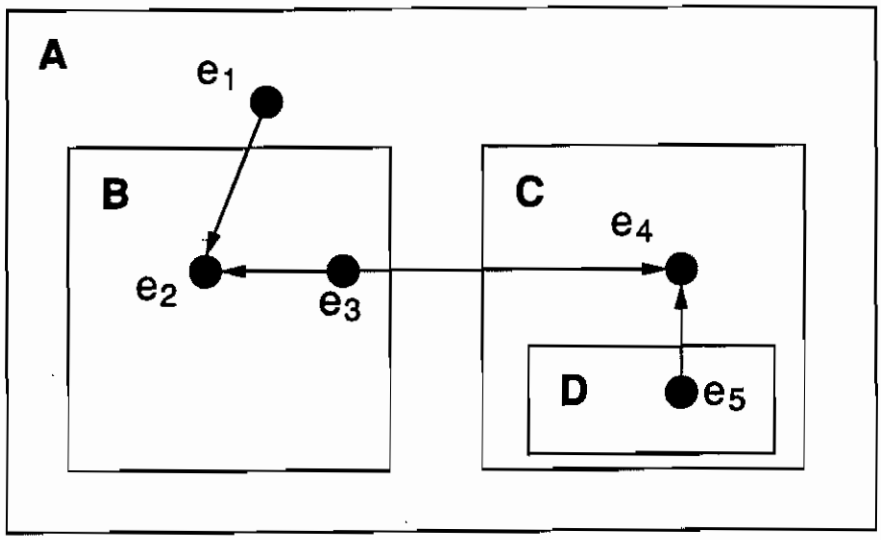

Figure 1: Problem domain

C: { e_4, $e_5 \longrightarrow e_4$ }

D: { e_5 } .

Constraints in a region can refer only to their own local plan. However, constraints in a region can be affected by changes in the plans of its subregions. For example, if an event e_6 is added to **B** and a constraint in **A** states that $e_1 \longrightarrow e_6$, then this constraint in **A** is violated because of a change in **B**. Moreover, changes in the local plan of a region can change the local plans of its subregions if relations are transitive. For example, suppose in Figure 1 that $e_3 \longrightarrow e_2$ was not present and $e_3 \longrightarrow e_1$ was added to the local plan of **A**. Because $\longrightarrow$ is transitive, $e_3 \longrightarrow e_2$ must be added to the local plan of **B**.[5] However, the search constraint localization requires that this should not affect the constraints of the subregion **B**.

How the search flows from one region to another in this example is illustrated in Figure 2 and is the subject of the next section.

[5]In GEMPLAN this is true because adding temporal relations never violates constraints; only adding events does. GEMPLAN's constraint algorithms make sure any execution will be correct, and adding $\longrightarrow$ limits only the possible executions.

3 Search Space

In this section we introduce the basic data structures and describe how the search operates on them. In the first section, we associate a *regional search tree* with each region in the problem domain. In the second section, we explain how an *incarnation* limits the search space within a regional search tree. Finally, we describe how *search fixes* account for branching.

3.1 Regional search tree

We associate a *regional search tree* with every region in the problem domain. Each node in this tree represents a unique regional plan of that region. Except for the root node of the tree, every node is derived from a father node through the application of a *search fix*. At this point, we make a distinction between two types of search fixes: a *local fix* and a *shift fix*. The local fix affects only the local plan of the father node to obtain a new son node. The shift fix affects the regional plan of the father node by constructing new subplans, i.e. by searching for new nodes in the subregional search trees; these nodes then become *subnodes* of the father node to obtain the son node. This implies that the search has direct *access* only to the region of the current search node and to its subregions. In a given node, there can be at most one subnode for every subregion.

Figure 2 represents these notions graphically and refers to the domain example of Figure 1. In the regional search tree of **A**, a_1 and a_3 are obtained from the root node a_0 via local fixes L_1 and L_2. The node a_2 is derived via a shift fix S_1. S_1 shifts control to the subregional trees of **B** and **C**. In **B** we find b_1 via the local fix L_3, and in **C** we find c_1 via the shift fix S_2; both b_1 and c_1 become subnodes of a_3, represented as $a_3(b_1, c_1)$. c_1 contains d_1 as its subnode, obtained in the subregion **D** of **C** via the local fix L_4.

In general, at any node, many alternative fixes will be available which account for the branching in the regional search trees. Because of this branching, a shift fix can be tried over and over until the corresponding subregional trees are exhausted. How this process can be controlled is the subject of the next section.

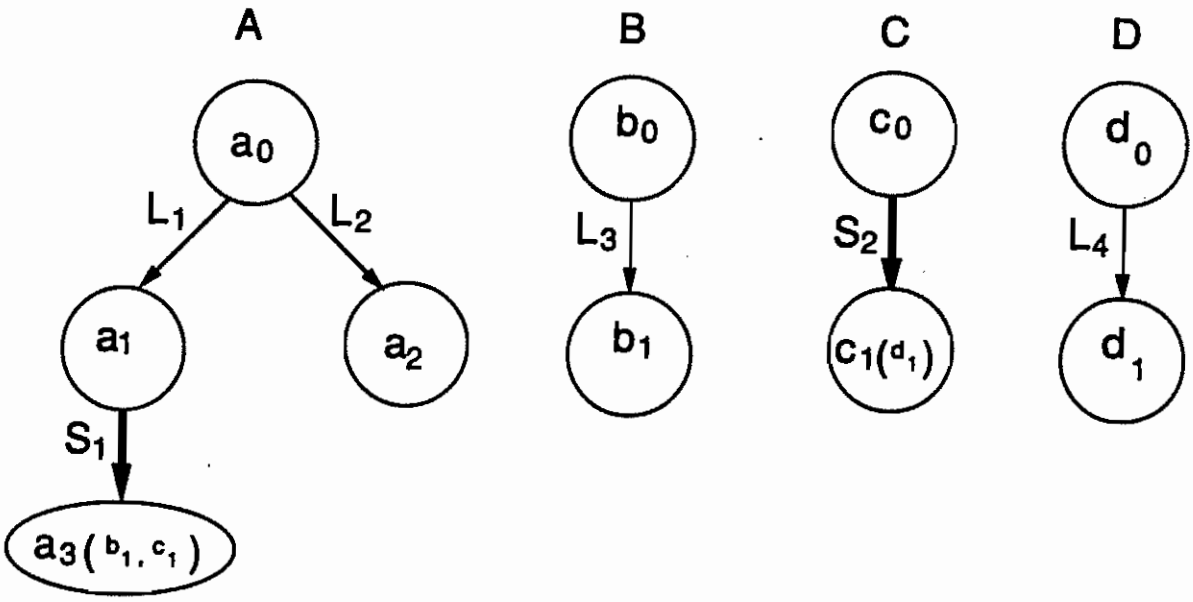

Figure 2: Regional search trees

3.2 Incarnation

As we have seen in Section 2, a plan consists of events and relations among these events. Plans are the objects upon which GEM-fixes and -constraints act. A local search fix corresponds to a *local GEM-fix*, and a shift search fix corresponds to a *foreign GEM-fix*. We use the following properties of GEM-fixes:

> A local GEM-fix changes only the local plan of a region.
>
> A foreign GEM-fix can change the local plan and the local subplans of a region.

A simple situation is depicted in Figure 3, and its corresponding search trees in Figure 4. In the global region **A**, we perform a local fix L_1 to obtain a_1. At that node, a shift fix S_1 triggers the search in **B**. However, before the search in region **B** can start, a root node b_0 is *generated*. Hence, this operation is called a *generate operation*. The plan of b_0 corresponds to the initial local plan of **B**. The information to generate a root node for **B** is contained in the foreign GEM-fix that corresponds to S_1. In **B**, a local fix produces b_1. The search stops at b_1 and control is returned to the regional

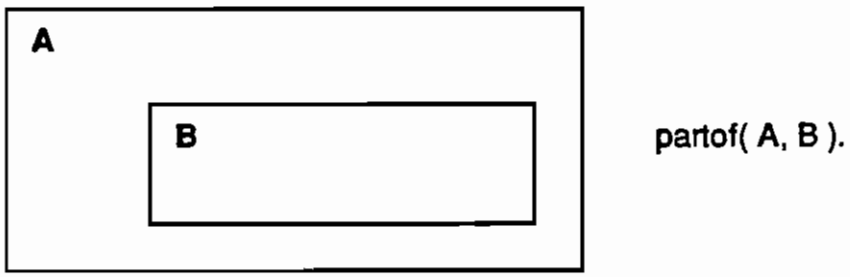

Figure 3: Simple domain

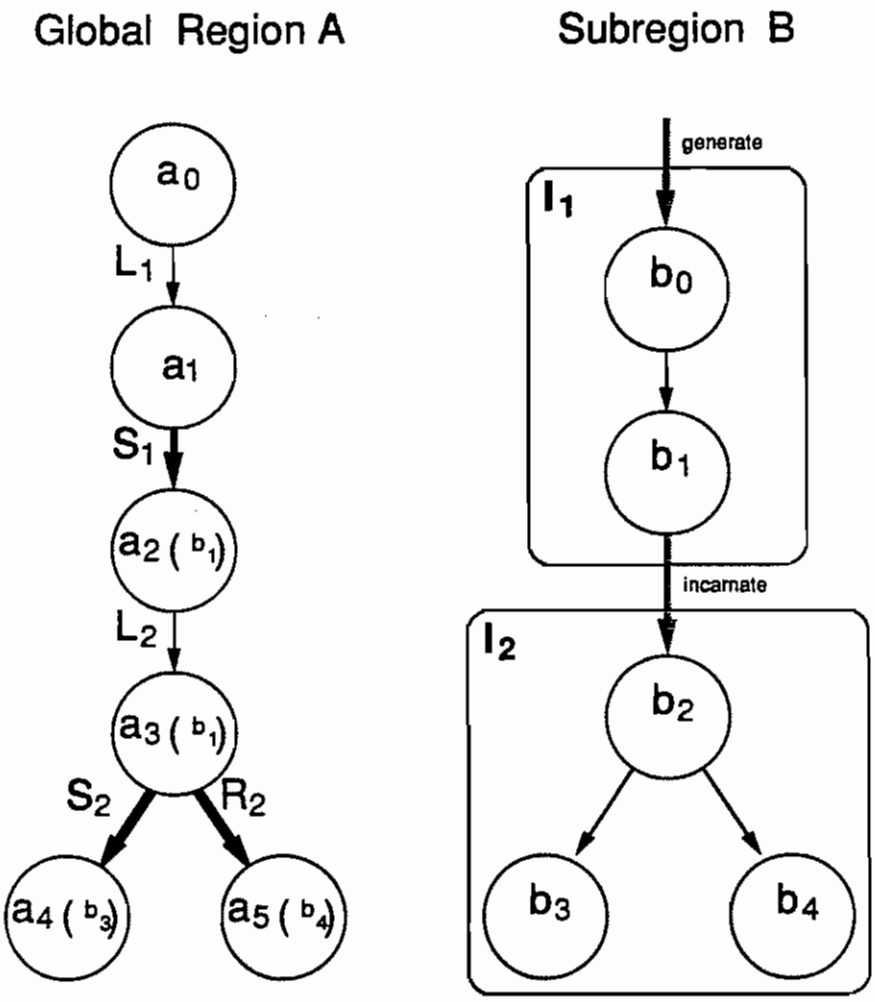

Figure 4: Incarnations of a regional search tree

tree of **A**.[6] In **A**, S_1 is complete and results in a_2 with b_1 as its subnode. At a_2 a local fix produces a_3; b_1 remains the subnode of a_3. At a_3, the shift fix S_2 tries to find a new subnode in **B**. However, before the search can restart in **B**, a new son node b_2 is derived from b_1, possibly by adding new events and relations to the local plan of b_1. This derivation is called an *incarnate operation*, as opposed to the *generate operation*; subregion **B** is reincarnated as b_2. The information to perform an incarnate operation is contained in the foreign GEM-fix that corresponds to S_2. At b_2, a local fix gives b_3. In **A**, S_2 is completed and results in a_4 with a new subnode b_3.

As this example shows, every time a generate or an incarnate operation is performed, a new subtree is created. Such a subtree of a regional search tree is called an *incarnation* of the region. In **B**, we have two incarnations, I_1 and I_2, which have as root nodes respectively b_0 and b_2.[7]

The partitioning of a regional search tree into incarnations reduces the search space when a shift fix is retried. When a shift fix is retried, it only makes sense to search for an alternative solution node within the same incarnation that was created when the shift fix was applied for the first time. We call this first application of a shift search fix *a new shift search fix* and call any subsequent retrial *a retry shift search fix.* If, as is shown in Figure 4, control goes back to a_3, and S_2 is retried to find an alternative subplan for region **B**, indicated by R_2, then the search in **B** for a new node b_4 is constrained to the same incarnation I_2 that was initiated by the new shift fix S_2. The global region **A** has only one incarnation which equals the whole search tree; we call this incarnation the *global incarnation.*

A new shift fix can generate and incarnate multiple incarnations for different subregions. In Section 5, we explain the details of applying and retrying such complex shift fixes.

3.3 Branching

The first source of branching in a regional search tree follows from the shift fixes which can be retried. However, retry shift fixes were possible only as a result of branching itself, and hence there must be a basic reason for branching. Every node has a unique

[6] Why the search stops at b_1 is not important in this discussion and will be explained much later in Section 6.3.

[7] In Section 4, we will see that the incarnations do not completely partition all of the nodes of the search space because some nodes do not belong to any incarnation.

plan, and for that plan any of the regional constraints can be checked. This checking discovers bugs in the plan. These bugs trigger GEM-fixes that might be tried to solve these bugs. There can be multiple fixes for each constraint as well as multiple ways to apply the same fix.

4 Overlap

In this section we discuss the problem of overlapping regions, and describe how this problem complicates the search considerably. In the first section, we define *overlap*. In the second section, we explain how overlap demands a special mechanism to maintain consistency. In Section 4.3, we show there is a trade-off between consistency and satisfaction. Finally, we introduce a data structure to represent consistency.

4.1 Definition

Overlap between two regions is defined as follows:

```
overlap( R1, R2 ):- t_partof( R1, Shared ),
                    t_partof( R2, Shared ),
                    R1 \== R2 .
```

In words, two different regions R1 and R2 overlap if there is a region, Shared, at a level lower than both R1 and R2. The region Shared is called a *shared region*.

In the regional structure of Figure 5, the partof/2 relation determines that region **D** is a subregion of both **B** and **C** and therefore is shared by **B** and **C**. As a result, every plan in **D** is a subplan of both **B** and **C**.

4.2 Consistency

A search node is *inconsistent* if (1) it has different lower-level nodes for the same region or (2) if it has a lower level node for a shared region that is not present on all paths from the given node to that shared region; these paths are constructed via the subnodes with the partof/2 relation. If there is no overlap then all nodes are guaranteed to be consistent. However, in the case of overlap, a special mechanism is needed to maintain consistency.

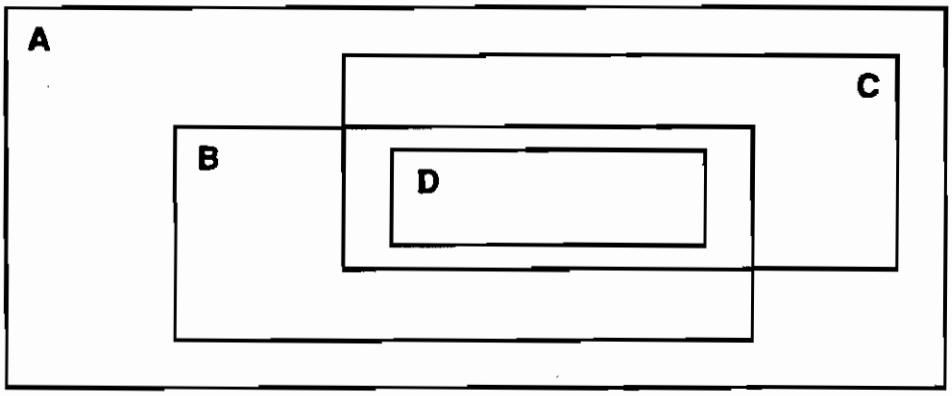

Figure 5: Domain with overlapping regions

Figure 6 represents part of the search trees for Figure 5. At a certain point during the search, we have node a_1 in **A**, b_1 in **B**, c_1 in **C**, and d_1 in **D**, where d_1 is a subnode of both b_1 and c_1, and b_1 and c_1 are subnodes of a_1. **A** performs a shift fix S_1 to **B** creating a new incarnation with root node $b_2(d_1)$. At b_2, a shift fix S_2 to **D** is performed, creating a new incarnation for **D** with root node d_2. Searching this incarnation gives d_3. This results in an new node b_3 with d_3 as its subnode. Control goes back to **A** to complete S_1. Subnode b_1 is replaced by b_3 found in **B**. However, $c_1(d_1)$ can no longer be included as a subnode of a_2, otherwise a_2 would have two different lower-level nodes, d_1 and d_3, for the shared region **D**. Therefore, c_2 is added to **C** and, together with b_3, is included as as subnode of a_2. The construction of c_2 is called a *completion operation*, and we associate a *complete search fix* C_1 with it. This completion forces the search to follow consistent paths if the search shifts from **A** to **C** and from **C** to **D**. Thus, after a shift fix is performed at a node, new subnodes explicitly calculated by the shift fix are added to the node, and the other subnodes of the node are completed.

Moreover, a completion operation can even introduce new subnodes. Figure 7 and its search trees in Figure 8 illustrate this. **C** is a subregion of **B** but also of **A**, and **B** is subregion of **A**. Hence, **C** is a region shared by **A** and **B**, and therefore **A** and **B** overlap. **A** performs a shift fix S_1 to **B**, and **B** performs a shift fix S_2 to **C**. In **C**, c_1 is found, and in **B**, $b_1(c_1)$ is found. To obtain a_1 in **A**, we include the explicitly calculated subnode b_1, but completion also must include the indirectly created subnode c_1 as a subnode of a_1. In this case, no complete search fix is performed during completion. The mechanism described implies that a completion operation will create at most one complete fix in every subregion.

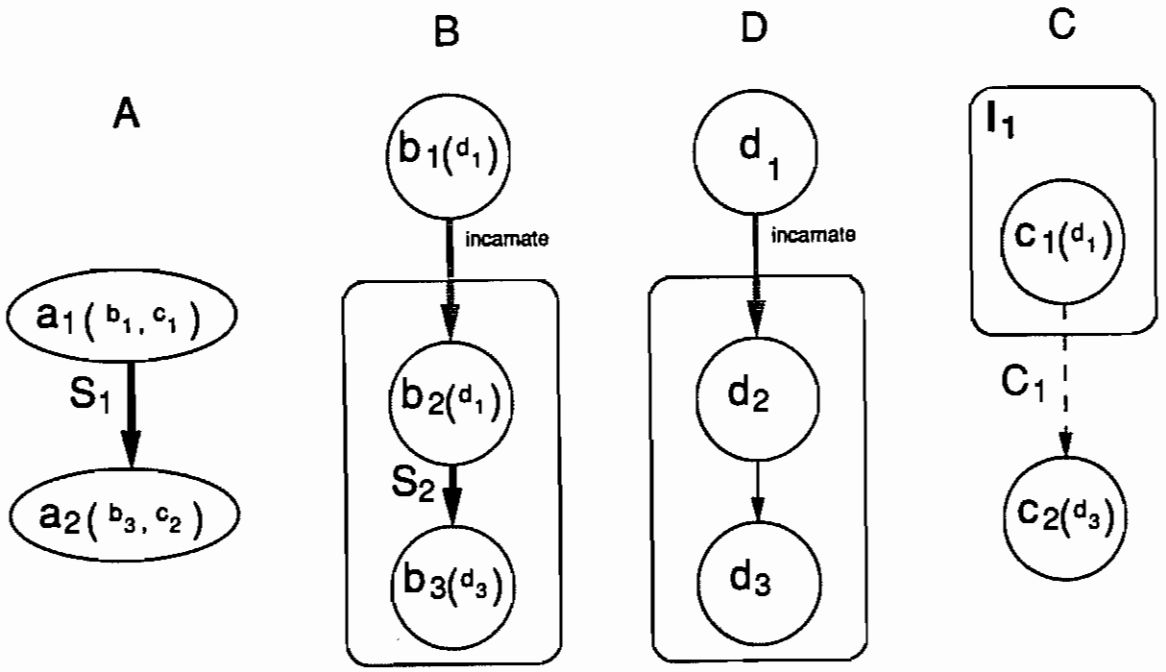

Figure 6: Search trees for domain with overlapping regions

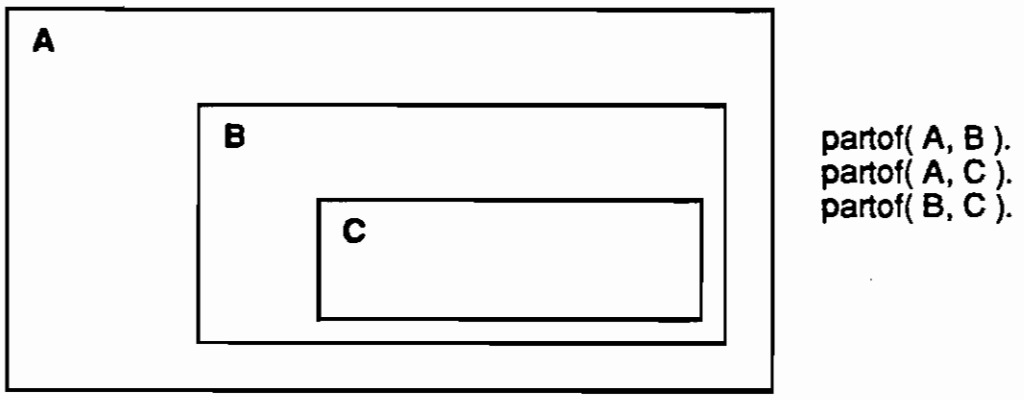

Figure 7: New subnodes by completion

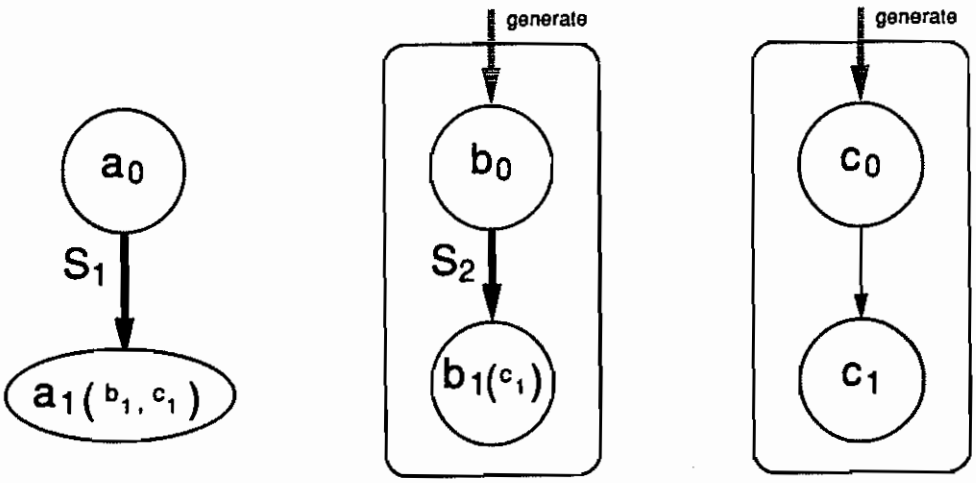

Figure 8: Search trees

The son nodes that result from complete fixes do not belong to any incarnation, and hence they will never be selected as search nodes, as we can explain with the example in Figure 6. There are only two ways to shift control back to region C: via a retry shift fix of S_1 at a_1 or via a new shift fix at a_2. In the case of a retry, the previous application of S_1 is no longer valid, and hence the complete fix C_1 that originated from S_1 is no longer valid. Therefore, c_2 should not belong to incarnation I_1. In the case of a new shift fix, a new incarnation will be created, together with a root node for that incarnation. That root node will have c_2 as its father node, but c_2 should not be included in this new incarnation.

Finally, completion operations are performed during the application of a shift fix that generates and incarnates multiple regions, some of which can be shared by others. We postpone the explanation of this mechanism to Section 5.

4.3 Consistency vs. satisfaction

So far, we have not explained why the search stops at a node in an incarnation. We shall describe that in full detail in Section 6.3. One obvious criterion is to stop when a *node is satisfied.* A node is satisfied if two conditions hold:

1. The local plan of that node satisfies the regional constraints

2. All of its subnodes are satisfied .

The search constraint localization property suggests that if shift fixes produce satisfied subnodes, then a node is satisfied if its own constraints are satisfied. Unfortunately, the condition of this statement does not hold in general; it holds only if there is no regional overlap, that is, if the regional structure is hierarchical. In Figure 6, the plan of node a_2 might satisfy all constraints of **A**, but it may well be that the completed node c_2 is no longer satisfied. The replacement of the subnode d_1 by subnode d_3 in c_1 can introduce *bugs* in the regional constraints of C. The definition of a local plan in Section 2.1 implies that regional constraints can refer only to events and relations in the local plan of a region. However, relations of a region can hold among events of that region and its subregions, and among events of different subregions. Therefore, the replacement or addition of a subnode in a given node, can introduce bugs in the local plan of that node. For example, suppose that d_3 contains an event *get*[8] that is not contained in d_1. Suppose the local plan of c_1 contains an event *provide*, and that there is a constraint in C that states that *provide* must happen before *get*. This constraint is satisfied in $c_1(d_1)$ but violated in $c_2(d_3)$ because there is no temporal relation in the local plan of c_2 that orders the event *provide* before *get*. As a result, the shift fix S_1 at a_1 introduces a subnode c_2 in a_2 that is not satisfied. If there were no overlap, then all subnodes produced by shift fixes would be satisfied. However, in the case of overlap, subnodes may in general be inconsistent and constraint satisfaction cannot be assured. Satisfaction is fairly easy to deal with, and therefore completion is always performed so that consistency is maintained throughout the search. A straightforward mechanism to deal with this satisfaction problem will be explained in Section 7.1.

4.4 Shared nodes

In this section we introduce a representation for consistency of a node. We define the *shared nodes* of a node as the set of its shared subnodes together with the shared nodes of its subnodes. If a node has no subnodes, then its shared nodes are empty. Every time a shift fix is performed, either a new shift fix or a retry fix, a new set of shared nodes is obtained. If this set differs from the shared nodes of the father, the new son node must be completed. Moreover, the difference between the old and the new set of shared nodes provides the necessary information on how the completion has to be

[8]In general, events are parameterized with times and objects.

performed. A set of shared nodes can contain at the most one node for every shared region.

In Figure 6, node a_1 will have $\{d_1\}$ as its set of shared nodes. During the application of S_1, this set will change to $\{d_3\}$. Comparing these two sets tells us that the shared node d_1 changed to d_3. Since d_1 is not a subnode of a_1, d_1 must be a shared node of one of its subnodes. This can be determined statically from the domain structure (see also 7.2.2). In this case, we find that d_1 is shared by c_1, and therefore the completion must be applied recursively to the subnode c_1 with respect to the change $d_1 \rightsquigarrow d_3$. At c_1 we find that d_1 is a subnode, and therefore the complete fix C_1 replaces it by d_3. In Figure 8 the situation is a little different. The set of shared nodes of a_0 is empty. During the application of S_1, c_1 is added to the set of shared nodes. Now, the comparison shows that c_1 is a new shared node of a_1. Since C is a subregion of **A**, c_1 is added as a new subnode of a_1, which performs the completion in one step.

From the examples we derive the following completion algorithm. First, we replace or add the subnodes that were explicitly calculated by the shift fix: in Figure 6, b_1 is such a subnode. The application of a shift fix guarantees that these subnodes are consistent and hence that they need no further attention for completion. A shift fix will always transform the set of shared nodes at a given node, into a new set. This new set of shared nodes will also be consistent. We compare these two sets and obtain a set of replacements and a set of new shared nodes. Every member of these sets has a unique region associated with it. No two members have the same region. Next, we take the replacements and new shared nodes whose region is a subregion of the given node, and respectively replace or add them as subnodes. Finally, we complete the remaining subnodes, i.e. the subnodes that were not added or replaced, with respect to the set of replacements and the set of new shared nodes. We cannot remove the members from these sets that were already replaced or added as subnodes because partof/2 defines a partial order and not a total order. As a result, they might still be needed in the completion at lower levels. The completion of the remaining subnodes is done recursively and adds complete fixes. If there are no remaining subnodes, the completion stops.

5 Shift fixes

The shift fixes are the most complex search fixes. In this section we study the semantics of shift fixes and their application.

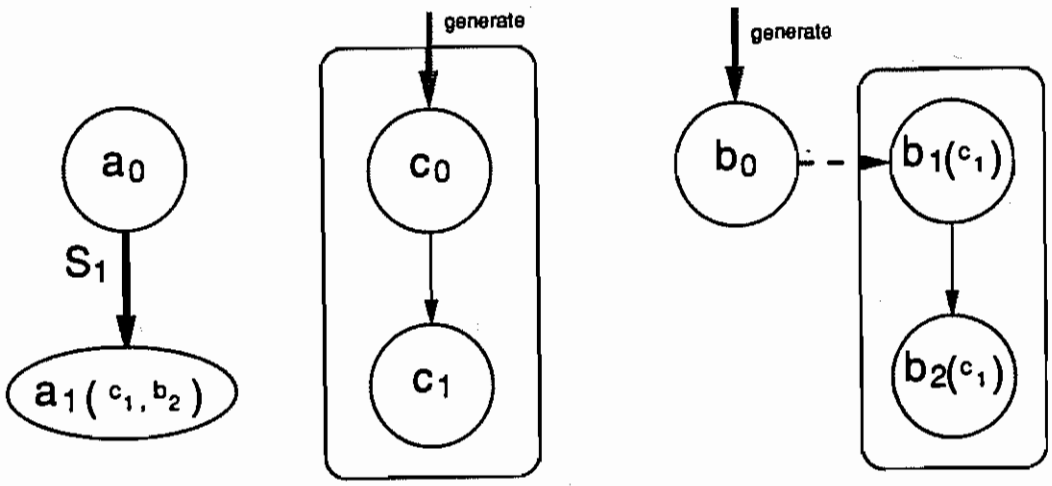

Figure 9: Search trees

5.1 Order of subregions

As we saw in Section 3, there are two types of shift fixes: the new shift fix and the retry shift fix. In general, both shift control to multiple subregional search trees. The order in which these subregional trees are explored is important. We illustrate this in Figure 9, which corresponds to the domain in Figure 7. At a_1 we perform a new shift fix S_1 to both **B** and **C**. This is possible, since both **B** and **C** are subregions of **A**. **C** is explored first, c_0 is generated, and some local searching results in c_1. Next, we explore **B**. The root node b_0 is generated. b_0 is not consistent as a result of the previous search in its subregion **C**. Therefore, b_0 is replaced by a new root node $b_1(c_1)$ that contains c_1 as its subnode.[9] Local search in **B** gives b_2. In **A**, S_1 results in a_1 without a completion operation. In Figure 10, subregions are visited in the reverse order, first **B** and then **C**. First, from the generated root node b_0, we perform a local fix and obtain b_1'. Next, we search **C** and obtain c_1. As explained in Section 4.2, before a_1' is constructed, a completion operation is performed which applied a complete fix at b_1' to obtain $b_2'(c_1)$. In general, the plan constructed via constraint satisfaction at b_2' will be different from that at b_2, and hence a_1 will be different from a_1'. This shows that the order in which the subregions of a shift fix are visited must be specified to determine its precise meaning.

[9]This completion operation during the application of a shift fix was mentioned at the end of Section 4.2.

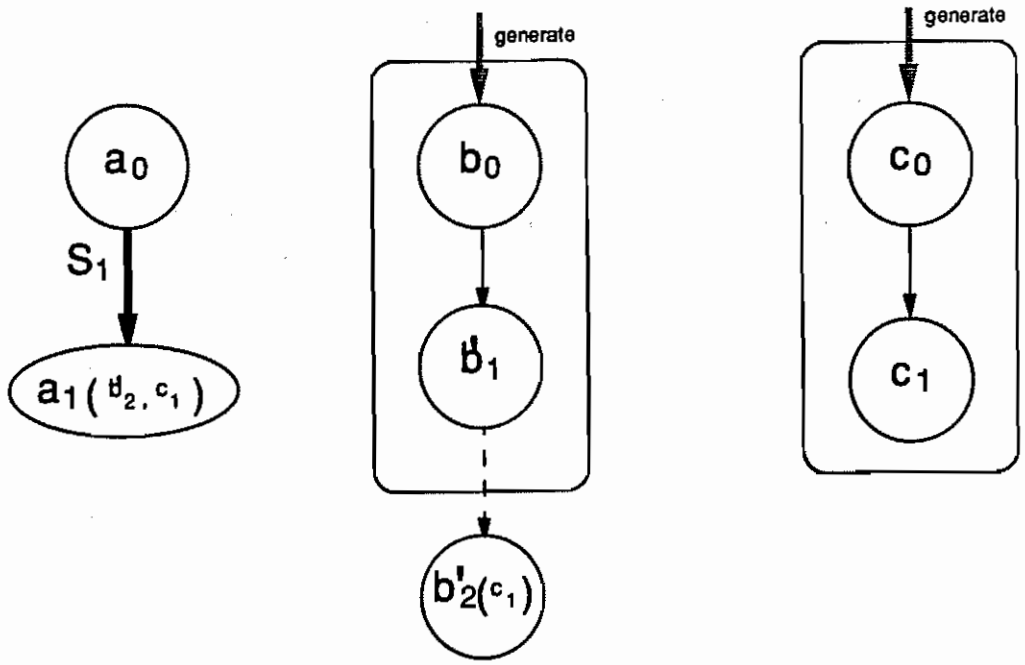

Figure 10: Search trees

From the above examples, we learn that, in general, completion is performed at two places: first, before we start the search in a new incarnation, we complete the root node of this incarnation; second, after all of the subregions have been visited, we complete all of the subnodes that were found. In general, completion can recursively affect lower-level search nodes.

5.2 Application of a shift fix

In Section 4.4 we saw that consistency is maintained with respect to a set of shared nodes. Every search node has its set of shared nodes. When a new shift is performed at a given node, we start with the shared nodes of that node. During the shift, we incrementally update this set of shared nodes as we go from one incarnation to the next, and use this current set of shared nodes to complete the root node. After all of the subregions have been searched, the subnodes are completed with respect to the final set of shared nodes.[10] These final shared nodes become the new shared nodes of

[10]The completion algorithm explained in Section 4.4 was based on the assumption that the calculated solution subnodes were consistent with respect to the new set of shared nodes.

the new son node that resulted from the new shift fix.

The application of a retry fix is more involved. A retry fix exists because the incarnations created by a (new) shift fix can be explored in multiple ways to find different solutions. Every new application of a retry fix will further explore these incarnations. Hence, every retry fix will start from a previous *state* of these incarnations, obtained by the last shift fix. As we saw in the previous section, these incarnations cannot be searched independently. Let us denote a sequence of incarnations as $I_1, \ldots, I_n$. These incarnations correspond to the different subregions that were visited by the shift fix. In general, the incarnation I_k depends upon the sequence $I_1, \ldots, I_{k-1}$. Therefore, if a retry fix is applied starting from the sequence of incarnations obtained by the previous shift fix, then we will first explore I_n, keeping the solutions already found in $I_1, \ldots, I_{n-1}$. If a new solution is found, I_n will be in new state I'_n, and the sequence $I_1, \ldots, I_{n-1}, I'_n$ will be kept for the next application of a retry fix. However, if I_n is exhausted, and no new solution is found, we reinstall the initial incarnation that was incarnated or generated by the corresponding new shift fix, call it I^0_n, and explore I_{n-1}. In general, a shift fix, either a retry shift fix or a new shift fix, will be in a state of execution represented by the sequence $I_1, \ldots, I_k, I^0_{k+1}, \ldots, I^0_n$. I_k is the incarnation currently being searched; either a solution is found, in which case we go forward to I^0_{k+1}, or I_k is exhausted, in which case we reinstall I^0_k and go back to I_{k-1}. If $k = n$ and we must go forward, then we find a solution for the shift fix; if $k = 1$ and we must go back, then the shift fix fails.

This application mechanism for shift fixes performs a depth-first, left-to-right search in the complete search space of incarnations for the subregions in a shift fix. As a result, it is enough to keep the information about the last applied shift fix. The following information is stored at the node at which the shift fix can be retried. The initial incarnations I^0_k and their root nodes r_k are stored. These were incarnated or generated by the shift fix, and are independent of each other; hence, this information will remain the same for every application of a retry fix. The incarnations I_k and their solution nodes s_k are stored. These were found by the previous application of the shift fix. We call such an incarnation a *incarnation state*. Finally, the set of shared nodes SN_k is stored. This was the current set of shared nodes when I^0_k was explored the last time. Whenever we have to go back to I_k, we reinstall SN_k as the current set of shared nodes. These three data structures together form a context: $context(I^0_k(r_k), I_k(s_k), SN_k)$. Thus, what we keep at a node where a shift fix can be retried is a sequence of contexts.

6 Implementation

In this section, we describe the implementation of the localized search. The implementation is in Quintus Prolog and uses Quintus's module facility. We will specify the modules and highlight some important implementation aspects. We assume that the reader has an understanding of logic programming. This section is meant as a basic explanation for those who want to integrate this search algorithm in their system and tune the search. In addition, it gives an introduction to those who want to extend and maintain the code, but eventually, these people will have to study the complete code and its comments.

Figure 11 gives the dependency diagram of the modules. All data types are implemented with terms: we make no use of the Prolog data base to represent data, and hence operations on data types never cause side effects.

Search_main is the highest-level procedural module and implements the search algorithm. This algorithm is generic, and a considerable amount of particular control and heuristic tuning of the search is obtained from the predicates defined by the user in *search_userctl*. The procedural modules *search_main* and *search_userctl* operate on different data types: *search_node* implements regional search trees, *search_incarnation* implements a search incarnation which groups nodes of a regional search tree, and *search_fix* abstracts the different search fixes that cause branching in the search trees. *Gem* represents a collection of procedures and data types.[11] First, it checks constraints and applies methods to solve unsatisfied constraints in a plan. Second, it implements GEM-fixes, GEM-constraints, plans, and the regional description of the problem. *Gem* will not be explained in depth, because it is not really part of the search. *Search_userctl* uses *gem* to construct the search fixes and check the constraints, and *gem* asks questions to *search_node* about the search space. *Gem_fix_constraint* defines the interface between the search and *gem*. *Interface* represents a group of modules that define a graphical interface between the user and the entire system. Finally, there are some lower-level modules that are not represented in Figure 11, like queues, association lists, etc. and which will not be discussed in this section.

[11] We do not capitalize *gem* to distinguish it from the GEM description language. *Gem* implements the GEM-constraints and -fixes, the plan representation and the problem domain. However, *gem* is not part of the search. In fact, it already existed before the present search algorithm was implemented. *Search_main* and *search_userctl* make calls to functions of *gem* to modify plans.

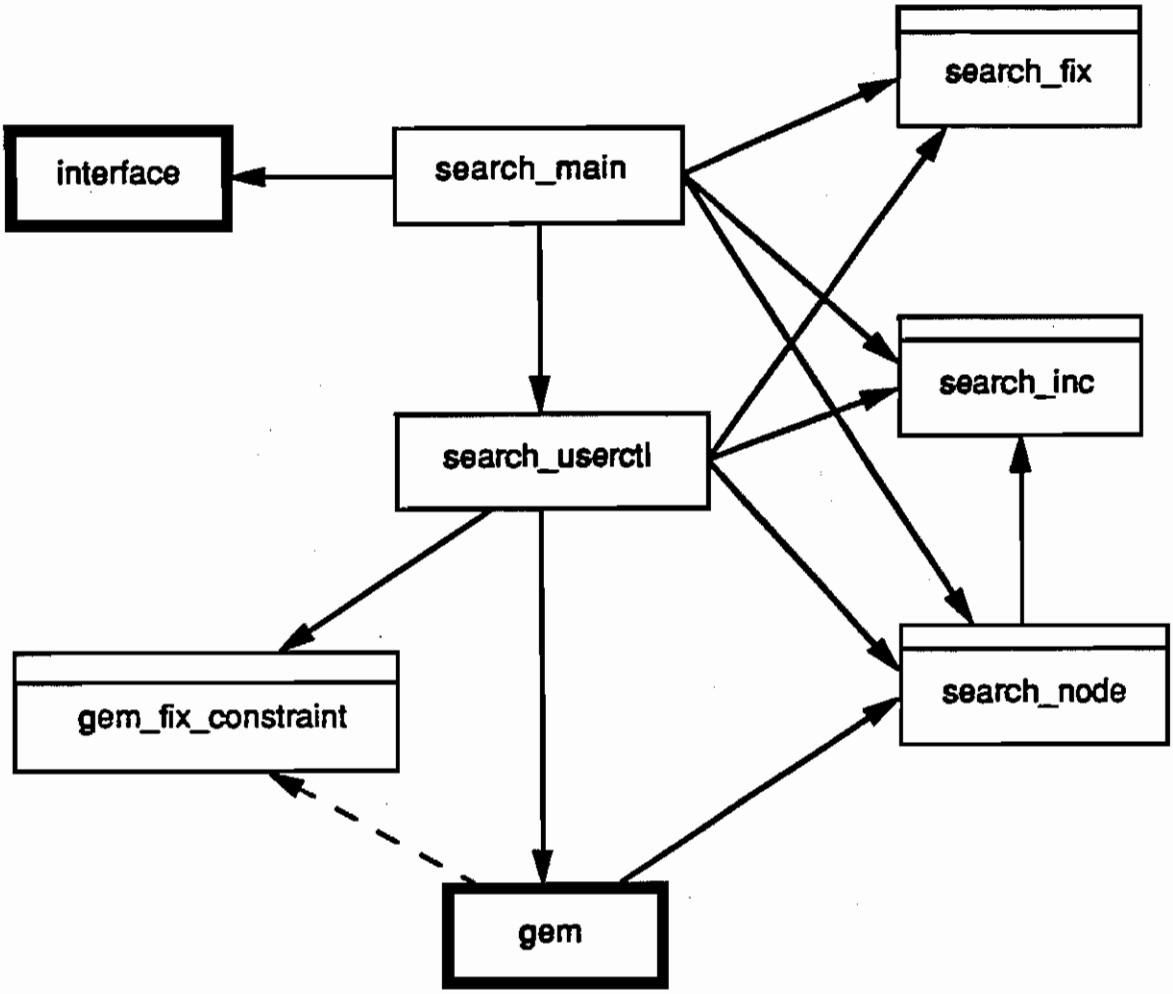

Figure 11: Module dependency diagram

6.1 search_main

This procedural module implements the search control. For that it uses other data modules and the procedural module *search_userctl*. In this section, we outline the high-level search algorithm.

6.1.1 search_incarnation

As explained in Section 3.2, the regional search at one level is limited within a given incarnation of that region. We initialize the search, by setting up an initial incarnation for the global region. The global region is the highest-level region of the problem domain at hand. There will never be a shift fix to this global region, and hence, the global incarnation will coincide with the complete global regional searchtree. All other incarnations are created during the search, and overall, they constitute a subset of their regional searchtree. One search step is performed at an incarnation as follows[12]:

```
:- use_module('search_incarnation.pl', [
        expunge_successors/3,
        put_node_inc/3 ]).

:- use_module('search_node.pl', [ void_node/1 ]).

:- use_module('search_userctl.pl', [
        prepare_fix/3,
        postpone_node/1,
        put_and_select_node/4,
        prune_node/1,
        satisfied_node/1,
        select_node/3 ]).

:- mode search_incarnation( +, +, -, - ).
search_incarnation( Inc, Node, Inc, Node ):-
        satisfied_node( Node ), !.
search_incarnation( Inc, Node, NewInc, NewNode ):-
        postpone_node( Node, NewNode ), !,
```

[12]The definitions presented here are slightly different from the actual code defined in *search_main* because we eliminated the calls to the *interface* that perform tracing. Also, we mention only the imported predicates used in the predicate definitions being discussed.

```
        put_node_inc( NewNode, Inc, NewInc ).
search_incarnation( Inc, Node, NewInc, NewNode ):-
        prune_node( Node ), !,
        expunge_successors( Inc, Node, ExpInc ),
        select_node( ExpInc, NextNode, NextInc ),
        search_incarnation( NextInc, NextNode, NewInc, NewNode ).
search_incarnation( Inc, Node, NewInc, SolNode ):-
        prepare_fix( Node, Fix, FixNode ),
        apply_fix( Fix, FixNode, Father, Son ),
        son_or_father( Inc, Father, Son, FixInc, PutNode ),
        put_and_select_node( PutNode, FixInc, NextNode, NextInc ),
        search_incarnation( NextInc, NextNode, NewInc, SolNode ).
```

Intuitively, the predicate search_incarnation(Inc+, Node+, NewInc−, NewNode−) tries to expand the regional searchtree starting from the node Node. The search is performed within a given incarnation starting from a state Inc. Inc is a collection of search nodes already visited by the search. The data type of Inc will be explained in Section 6.4. Search_incarnation/4 will add at most one node to Inc, and it can remove several nodes from Inc to obtain NewInc; NewInc represents the new state of the search incarnation after the search has been performed. Logically, Node is part of Inc, and NewNode is part of NewInc, but, for efficiency, Node does not belong to the incarnation Inc. NewNode is called a *solution node* found in an incarnation that changed its state from Inc to NewInc. This predicate fails if no NewNode can be found.

The first two clauses are not recursive. They catch the simple cases corresponding to the following conditions:

satisfied_node/1 Node is satisfied. However, this does not necessarily mean that the regional plan of that node is satisfied, as we explained in 4.3. The search can never be continued from this node within the same incarnation, and hence it is not added to Inc.

postpone_node/2 We postpone the search in the given incarnation at Node, and replace it by NewNode. Later, the search should be able to continue from NewNode, and hence we add it to Inc to obtain NewInc. NewNode may not be equal to Node, so the search will not be postponed forever at Node in Inc.

The third clause prunes a Node from the regional searchtree. This means that the whole subtree with root Node will not be considered any more. Therefore Node is not added to

Inc, and all successor nodes of Node are removed from Inc to obtain ExpInc. Then, we select another Node from ExpInc, NextNode, and continue the search from NextNode within NextInc. NextInc is equal to ExpInc without node NextNode: If no Node can be selected from ExpInc, e.g. because ExpInc is exhausted, then select_node/3 fails, and hence search_incarnation/4 fails. This the only way in which search_incarnation/4 can fail.

Finally, the last clause is the non-trivial one. Prepare_fix/3 investigates Node and produces a search fix Fix. It also changes Node to FixNode to reflect the investigation. Prepare_fix/3 is specified in Section 6.3.1. Apply_fix/5 is studied in detail in Subsection 6.1.2: It applies Fix, which changes FixNode to Father, and produces Son, the son node of Father. The predicate son_or_father/5 is defined as

```
son_or_father( Inc, Node, Son, Inc, Node ):-
        void_node( Son ), !. % RED CUT
son_or_father( Inc, Father, Son, NewInc, Son ):-
        put_node_inc( Father, Inc, NewInc ).
```

If Son is a void node, it is discarded, and we select Node. In that case, Inc doesn't change. The second clause puts Father in Inc to obtain NewInc, and returns Son. The call to put_and_select_node/4 selects the node from which the search has to continue: in the case of a depth-first search strategy within an incarnation, the last visited node FixNode is selected as NextNode, in which case NextInc equals FixInc[13]; however, the search can continue from any node NextNode from FixInc, and therefore, NextInc equals FixInc to which FixNode is added and from which NextNode is deleted.

This high-level description of the search already reveals some important properties. First, search_incarnation/4 is deterministic: this follows from the cuts and from the determinism of its subgoals. The cuts are red[14], but not scarlet: the first call in the body of the first three clauses acts as a guard, and logically they are mutual exclusive; the last clause has no such guard, and therefore the cuts are red.[15] Determinism is necessary to have full control over the search, and to allow for a general search strategy. For example, the predicate prepare_fix/3 logically could be indeterminate, because multiple fixes can be prepared at a given node. If prepare_fix/3 were indeterminate,

[13]This is the current search strategy (see Section 6.4).

[14]*Red* cuts prune away solutions as well as proofs. *Green* cuts prune away proofs but no solutions.

[15]If the last call had a guard, the cuts would be green. Because we abstract the node representation, we cannot dispatch on the node parameter using Quintus's first parameter indexing scheme, and therefore green cuts would still be needed for efficiency.

it would not need a third parameter. If a fix failed, the search would backtrack to prepare_fix and try another one. (The reader can try to modify the search in that way. He will find that his algorithm degenerates into a depth-first search.) Second, search_incarnation/4 is generic, and comprises all possible search strategies within an incarnation. The particular strategy is determined by the imported predicates from *search_usercil*. At the node level, prepare_fix/3 determines which branch to chose. At the incarnation level, put_and_select_node/4, select_node/3, and prune_node/1 determine the flow of control among the nodes within an incarnation. Finally, at the interincarnation level, postpone_node/2 and satisfied_node/1 transfer control to the next-higher-level incarnation. How we can go to a lower-level incarnation will be explained in the next subsection.

6.1.2 apply_fix

The definition for apply_fix is as follows:

```
:- use_module('search_fix.pl', [
        fail_searchfix/1,
        local_searchfix/1,
        new_shift_searchfix/1,
        genandincs_of_searchfix/2,
        retry_shift_searchfix/1,
        retry_of_searchfix/2,
        void_searchfix/1 ]).

:- use_module('search_node.pl', [
        add_fail_branch/3,
        contexts_of_retry/2,
        sharednodes_of_context/2,
        init_inc_of_context/2,
        state_inc_of_context/2,
        sharednodes_of_node/2,
        void_node/1 ]).

apply_fix( Fix, Node, Father, Son ):-
        fail_searchfix( Fix ), !,
        void_node( Son ),
        add_fail_branch( Fix, Node, Father ).
```

```
apply_fix( Fix, Node, Node, Son):-
        void_searchfix( Fix ), !,
        void_node( Son ).
apply_fix( Fix, Node, Father, Son ):-
        local_searchfix( Fix ), !,
        local_branch( Fix, Node, Father, Son ).
apply_fix( Fix, Node, Father, Son ):-
        new_shift_searchfix( Fix ), !,
        genandincs_of_searchfix( Fix, GenAndIncs ),
        sharednodes_of_node( Node, SharedNodes ),
        make_incarnations( GenAndIncs, InitIncs ),
        check_incarnations( InitIncs, RevContexts, SharedNodes, NewSharedNodes ),
        shift_branch( Fix, Node, RevContexts, NewSharedNodes, Father, Son ).
apply_fix( Fix, Node, Father, Son ):-
        retry_shift_searchfix( Fix ), !,
        retry_of_searchfix( Fix, Retry ),
        contexts_of_retry( Retry, [LastContext| AccRevContexts] ),
        sharednodes_of_context( LastContext, SharedNodes ),
        init_inc_of_context( LastContext, LastInitInc ),
        state_inc_of_context( LastContext, LastStateInc ),
        acc_check_incs( LastStateInc, LastInitInc, [],
                        AccRevContexts, RevContexts,
                        SharedNodes, NewSharedNodes ),
        shift_branch( Fix, Node, RevContexts, NewSharedNodes, Father, Son ).
```

Apply_fix(Fix+, Node+, Father−, Son−) has the following meaning. The search fix Fix is applied to the search node Node. The input node Node changes to Father, and the application results in a child node Son of Father. The five clauses dispatch on the type of fix. These types are defined in *search_fix* (Section 6.5); they are mutually exclusive, and therefore the cuts are green. A very important invariant for the son node Son is that the plan of this node is consistent with its shared nodes (see definition of shared nodes of a node in Section 4.4). This guarantees that throughout the search, every node will be consistent with its own set of shared nodes.

The first clause corresponds to a failed fix. At first, it may seem strange that prepare_fix/3 has prepared a fail fix.[16] As we will see in Section 6.3, prepare_fix/3 is more

[16]In the actual implementation, prepare_fix has five parameters: two extra parameters are used for tracing and are not presented here.

involved than just selecting a fix at a given node: in general, a GEM-fix is chosen and applied to the plan, and a result search fix is returned. There is a clear distinction between GEM-fixes and search fixes: GEM-fixes are applied to a plan, whereas search fixes are applied to a search node. The application of a GEM-fix, which corresponds to a call to the procedure fix_bugs/5 from prepare_fix/3, is abstracted in *gem_fix_constraint* (Section 6.6) and is defined in *gem*. Fail_searchfix/1 corresponds to a failed GEM-fix on the plan of the given node Node. Father is obtained from Node by adding the fix as a fail branch. The son node Son is void.

The second clause corresponds to a void fix, which means that no fix has been prepared by prepare_fix/3. Again, this may seem strange at first, but it is not. As we will see in Section 6.3, prepare_fix/3 does not only return a search fix, it also changes the node from which the fix is prepared. A void search fix means that prepare_fix/3 has manipulated the node, but that no search fix has been extracted, e.g. only constraints have been checked at the node.[17] In this case, the search tree doesn't change, and a void son is returned.

The third clause catches a local fix. Fix contains all of the information necessary to calculate Father and Son and add them to the regional searchtree. The definition of local_branch/4 is straightforward and is not presented here.

The fourth clause applies a new shift fix. Shift fixes were explained in Section 5. GenAndIncs is the sequence of generate and incarnate operations that constitute the new shift fix prepared by prepare_fix/3. The elements of GenAndIncs are either of type *generate_foreign* or *incarnate_foreign* and are abstracted in *search_fix*. They contain all of the information to set up the corresponding initial incarnations to which a shift will be performed: make_incarnations(+, −) is fairly straightforward and not presented here. Check_incarnations/4 tries to find a sequence of solutions by searching the initial incarnations, InitIncs, using SharedNodes as the initial set of shared nodes used for completion. NewSharedNodes is the final set of shared nodes, and RevContexts is the reversed sequence of contexts of the incarnations to which a shift has been performed. This operation is rather involved and is the subject of the next section. Finally, a shift branch is added to the search tree: shift_branch/6 transforms Node to Father using Fix and RevContexts, which are both used to construct a retry shift fix. This retry fix is added to Father as an alternative fix that can be tried if Father is revisited. NewSharedNodes are the new shared nodes for Son, and the subnodes of Son are

[17]This is the case for the current implementation of prepare_fix/3. Constraints are checked via a call to check_constraints/5 from prepare_fix/3: check_constraints/5 is abstracted in *gem_fix_constraint* (Section 6.6) and defined in *gem*.

constructed from the subnodes of Father (or Node) together with the solution nodes found in the incarnation states of RevContexts. All of the subnodes are completed with respect to NewSharedNodes before they are added as the subnodes of Son. This completion operation and shift_branch/6, which calls this completion operation, are not presented here.

Finally, the fifth clause applies a retry shift fix. Contexts_of_retry/2, sharednodes_of_context/2, init_inc_of_context/2, and state_inc_of_context/2 extract the information from the retry fix necessary to start the search acc_check_incs/7. Acc_check_incs/7 tries to find a new sequence of contexts for the incarnations of the shift fix, as will be explained in the next subsection. Finally, a shift branch is added to the search tree, in the same way as for the new shift fix.

6.1.3 check_incarnations

As we saw in Section 5, a great deal of information is needed to execute a shift fix and to construct a retry fix from a previously applied shift fix. An initial incarnation, together with its root node, is represented in the data structure init_inc. A incarnation state, together with its solution node, is represented in state_inc. The data types init_inc and state_inc are abstracted in *search_node*. An init_inc and its corresponding state_inc, together with the shared nodes at this incarnation, form a context. The contexts of the incarnations of a shift fix form a component of a retry fix and are kept in reverse order because our depth-first strategy will recheck the last incarnation first. Context and retry are also abstracted in *search_node*.

```
:- use_module('search_node.pl', [
        sharednodes_of_context/2,
        init_inc_of_context/2,
        state_inc_of_context/2 ]).

check_incarnations( [FirstInitInc| RestInitIncs], RevContexts,
                    SharedNodes, NewSharedNodes ):-
        acc_check_incs( FirstInitInc, FirstInitInc, RestInitIncs,
                        [], RevContexts,
                        SharedNodes, NewSharedNodes ).

acc_check_incs( CurInc, CurInitInc, RestInitIncs,
                AccRevContexts, RevContexts,
```

```
                SharedNodes, NewSharedNodes ):-
        check_inc( CurInc, StateInc, SharedNodes, AccSharedNodes ), !,
        sharednodes_of_context( Context, SharedNodes ),
        init_inc_of_context( Context, CurInitInc ),
        state_inc_of_context( Context, StateInc ),
        cont_acc_check_incs( RestInitIncs, [Context| AccRevContexts], RevContexts,
                             AccSharedNodes, NewSharedNodes ).
acc_check_incs( _OldInc, OldInitInc, RestInitIncs,
                [NewContext| AccRevContexts], RevContexts,
                _OldSharedNodes, NewSharedNodes ):-
        !,
        sharednodes_of_context( NewContext, SharedNodes ),
        init_inc_of_context( NewContext, InitInc ),
        state_inc_of_context( NewContext, NewInc ),
        acc_check_incs( NewInc, InitInc, [OldInitInc| RestInitIncs],
                        AccRevContexts, RevContexts,
                        SharedNodes, NewSharedNodes ).
acc_check_incs( _FirstInc, _FirstInitInc, _RestInitIncs,
                [], fail_check,
                SharedNodes, SharedNodes ).
cont_acc_check_incs( [], RevContexts, RevContexts, SharedNodes, SharedNodes ).
cont_acc_check_incs( [NextInitInc| RestInitIncs], AccRevContexts, RevContexts,
                     SharedNodes, NewSharedNodes ):-
        acc_check_incs( NextInitInc, NextInitInc, RestInitIncs,
                        AccRevContexts, RevContexts,
                        SharedNodes, NewSharedNodes ).
```

The meaning of check_incarnations/4 follows from acc_check_incs/7, so we will explain only the latter.[18] The heading of acc_check_incs/7 is

```
acc_check_incs( CurInc+, CurInitInc+, RestInitIncs+,
                AccRevContexts+, RevContexts−, SharedNodes+, NewSharedNodes− )
```

CurInc is either a state_inc or an init_inc, and is the current incarnation under investigation. CurInitInc is the init_inc that corresponds to CurInc. RestInitIncs is a

[18]This specification is imprecise and incomplete. A rigorous specification would take several pages. The author believes this predicate can be better explained procedurally, rather than declaratively.

list of initial incarnations that remains to be checked in the order they appear in the list. The accumulating list parameter AccRevContexts is the sequence of contexts in reverse order constructed so far. SharedNodes is the accumulated set of shared nodes. The list RevContexts is the final sequence of contexts. The tail of RevContexts corresponds to AccRevContexts, but is not necessarily the same. This tail is preceded by a context corresponding to CurInitInc. The head list of RevContexts is constructed from the incarnations of RestInitInc in reverse order. If no such solution can be found for RevContexts, RevContexts will equal the constant fail_check.

The first clause tries to construct a (new) incarnation state StateInc from CurInc by calling check_inc/4. As we will see, check_inc/4 may need SharedNodes for completion, and transforms it into AccSharedNodes. Context is constructed, and accumulated in AccRevContexts. In cont_acc_check_incs/5, either there are no remaining initial incarnations, in which case a solution for RevContexts and SharedNodes is found, or the first element of the remaining initial incarnations is installed as the current incarnation under investigation. If check_inc/4 failed, the second clause of acc_check_incs/7 tries to backtrack by reinstalling the first context of AccRevContext as the current incarnation to be investigated. If no backtracking is possible, the third clause of acc_check_incs/7 returns fail_check as RevContexts.

Finally, let's look at the definition of check_inc/4[19].

```
:- use_module('search_node.pl', [
        sharednodes_of_node/2,
        merge_sharednodes/3,
        init_inc/1,
        state_inc/2,
        node_of_init_inc/2,
        incarnation_of_init_inc/2,
        node_of_state_inc/2,
        incarnation_of_state_inc/2 ]).

:- use_module('search_userctl.pl', [ select_node/1 ]).

check_inc( Inc, StateInc, SharedNodes, NewSharedNodes ):-
```

[19]The actual definition is more complicated than the one presented here because we make a distinction between incarnations that are initially satisfied and those that need to be checked. This mechanism is mentioned in Section 6.6. For this reason, init_inc/1, state_inc/2, and add_sharednode/3 do not actually exist.

```
        init_inc( Inc ), !,
        node_of_init_inc( Inc, Node ),
        complete_root_node( Node, SharedNodes, CompNode ),
        incarnation_of_init_inc( Inc, Incarnation ),
        search_incarnation( Incarnation, CompNode, NewIncarnation, NewNode ),
        node_of_state_inc( StateInc, NewNode ),
        incarnation_of_state_inc( StateInc, NewIncarnation ),
        sharednodes_of_node( NewNode, SubSharedNodes ),
        merge_sharednodes( SubSharedNodes, SharedNodes, TSharedNodes ),
        cond_include_sharednode( NewNode, TSharedNodes, NewSharedNodes ).
check_inc( Inc, StateInc, SharedNodes, NewSharedNodes ):-
        state_inc( Inc ), !,
        incarnation_of_state_inc( Inc, Incarnation ),
        select_node( Incarnation, Node, TempIncarnation ),
        search_incarnation( TempIncarnation, Node, NewIncarnation, NewNode ),
        node_of_state_inc( StateInc, NewNode ),
        incarnation_of_state_inc( StateInc, NewIncarnation ),
        sharednodes_of_node( NewNode, SubSharedNodes ),
        merge_sharednodes( SubSharedNodes, SharedNodes, TSharedNodes ),
        cond_include_sharednode( NewNode, TSharedNodes, NewSharedNodes ).
```

The definition dispatches on the first parameter Inc, which is either an initial incarnation or an incarnation state. In the first clause, the root node Node of Inc is completed with respect to SharedNodes to obtain CompNode. This completion will put nodes from SharedNodes into the subnodes of Node and complete the remaining subnodes of Node with respect to SharedNodes. This operation is the same as the completion performed in shift_branch/6. A recursive call to search_incarnation/4 tries to find a new solution NewNode. If it is successful, i.e. if NewNode is not a fail node, StateInc and NewSharedNodes are constructed. Merge_sharednodes/3 puts the new nodes of SubSharedNodes into SharedNodes, and NewNode is also included into NewSharedNodes if it is a shared node. The second clause is similar to the first, except that a search node is selected from the incarnation state, and therefore no completion is necessary.

6.1.4 Motivation

Why did we describe the high-level search control in such detail? First, in order to give a complete definition of search_incarnation/4, we had to refine to a level that includes all recursive calls to search_incarnation/4. Second, the search must be very well understood by someone who devises the module *search_userctl* for a particular application. For example, let us consider the shift fixes. The precise mechanism of check_incarnations and acc_check_incarnations must be well understood, not only to generate semantically correct fixes, but also to generate efficient fixes by taking the depth-first, left-to-right mechanism into account. We did not include the definitions for the completion mechanism, because they do not matter for *search_userctl*.

6.2 search_node

Search_node implements the fundamental data type of the search, a node of a regional searchtree. Nodes are created and modified during the search by *search_main*, modified and consulted by *search_userctl*, and consulted by *gem*. *Search_node* implements all node-related high-level predicates. The specification of these predicates is independent of the particular node representation. These predicates will not be described here but can be found in the declaration of the module itself.

The node term is defined as follows:

```
void_node( voidnode ).
id_of_node( voidnode, VoidId ):- void_id( VoidId ).
id_of_node( node(Id, _, _, _, _, _, _, _, _), Id ).
plan_of_node( node(_, Plan, _, _, _, _, _, _, _), Plan ).
expand_of_node( node(_, _, Expand, _, _, _, _, _, _), Expand ).
check_of_node( node(_, _, _, Check, _, _, _, _, _), Check ).
branches_of_node( node(_, _, _, _, Branches, _, _, _, _), Branches ).
father_of_node( node(_, _, _, _, _, Father, _, _, _), Father ).
subnodes_of_node( node(_, _, _, _, _, _, Subnodes, _, _), Subnodes ).
incarnation_of_node( node(_, _, _, _, _, _, _, IncId, _), IncId ).
sharednodes_of_node( node(_, _, _, _, _, _, _, _, SharedNodes), SharedNodes ).
```

6.2.1 id_of_node

Every node in the complete search space has a unique identifier associated with it. However, this identifier is never used by the search and is only for the sake of *gem*. The fact void_id/1 is defined in *search_constants*, which is not described in this report.

6.2.2 plan_of_node

Every node represents a unique plan. The actual plan representation is implemented by *gem*. In fact, the plan information is never used by the search, only by *gem*. However, the search will modify plans during the search, and therefore *gem* provides two functions:

```
% replace_subplan( Plan+, SubPlan+, NewPlan- )
% replace_or_put_subplan( Plan+, SubPlan+, NewPlan- ) .
```

Intuitively, NewPlan is obtained from Plan by replacing or putting SubPlan in it. However, *gem* represents plans using global data base facts, and as a result these functions cause side effects.

6.2.3 expand_of_node

This component contains all of the information upon which *search_userctl* bases its decision how to continue the search at that node. The expand term has two components:

```
fixbugs_of_expand( expand(FixBugs, _), FixBugs ).
retrys_of_expand( expand(_, Retrys), Retrys ).
```

Fixbugs is a list of procedures that are still untried to satisfy constraints for the plan of that node. The data type fixbug is a 4-term:

```
gemconstraint_of_fixbug( fixbug(GemConstraint, _, _, _), GemConstraint ).
gemfix_of_fixbug( fixbug(_, GemFix, _, _), GemFix ).
bugs_of_fixbug( fixbug(_, _, Bugs, _), Bugs ).
solutions_of_fixbug( fixbug(_, _, _, Solutions), Solutions ).
```

GemFix identifies a procedure to fix Bugs for the constraint GemConstraint. Solutions represents different ways by which the given GemFix procedure can be tried to solve

the bugs. The particular representation of these four parameters is part of *gem* and is unknown to the search. The interface between *gem* and the search is specified in *gem_fix_constraint*.

Retrys is the list of shift fixes that have already been performed at that node and for which alternative solutions might still be found. The data type retry is implemented as a 2-term:

```
creatorfix_of_retry( retry(Fix, _), Fix ).
contexts_of_retry( retry(_, Contexts), Contexts ).
```

The creator fix of a retry is the new shift fix that originated this retry fix. Contexts is a list with members of type context:

```
sharednodes_of_context( context(SharedNodes, _, _), SharedNodes ).
init_inc_of_context( context(_, InitInc, _), InitInc ).
state_inc_of_context( context(_, _, StateInc), StateInc ).
```

A context was described in Section 6.1.3. The particular term structure of the types init_inc and state_inc will not be described. It is enough to know that init_inc has a search incarnation and a search root node and that state_inc has a search incarnation and a solution search node. These components are selected by node_of_init_inc/2, incarnation_of_init_inc/2, node_of_state_inc/2, and incarnation_of_state_inc/2 in the definition of check_inc/4 (Section 6.1.3). Both the incarnations of a state_inc and an init_inc can only contain nodes that belong to a subregion of the given node, and because of the partial ordering among regions, no circularities will occur in these components of node.

6.2.4 check_of_node

Check_of_node has the following components:

```
satisfieds_of_check( check(Satisfieds, _, _, _), Satisfieds ).
visiteds_of_check( check(_, Visiteds, _, _), Visiteds ).
constraints_of_check( check(_, _, Constraints, _), Constraints ).
checkeds_of_check( check(_, _, _, Checkeds), Checkeds ).
```

Satisfieds, Visiteds, and Checkeds are lists of constraints, and Constraints is a queue of constraints.[20] All of these constraints are associated with the region of the node. Satisfieds are the constraints satisfied by the plan of the node; Visiteds are the constraints already checked, but not necessarily satisfied; Constraints is the queue of constraints waiting to be checked; and, finally, Checks are the constraints that were actually checked at this node. The particular representation of a constraint is part of *gem* and is unknown to the search. The interface between *gem* and the search is specified in *gem_fix_constraint*. The following invariant is true of every node: Satisfieds, Visiteds, and Constraints form a partition of the set of regional constraints. Checks is a subset of the union of Visiteds and Satisfieds; every constraint of Check that is satisfied must also belong to Satisfieds; and every constraint that has been checked at this node but is not yet satisfied must belong to Visiteds. However, not all constraints of Visiteds were actually checked at this node: some can be derived from the Visiteds component of the father node through the *partially affected* mechanism, which will not be described here. The component Checks, together with father_of_node (Section 6.2.6), is necessary to implement the following functions used by *gem*[21]:

```
% check_from_node( Node+, GemConstraint+, GemPlan- )
% check_before_node( Node+, GemConstraint+, GemPlan- )
% GemPlan is the plan of the closest ancestor node of Node at
% which GemConstraint was checked.
% If no such ancestor exists, then GemPlan is the empty list.
% The ancestors all belong to the same region, and
% hence it only makes sense if GemConstraint is a local constraint
% of the region to which Node belongs.
% In the case of check_from_node, Node is itself included in
% its ancestors; in the case of last_checked_before_node, it is not.
```

6.2.5 branches_of_node

Branches represents the methods that determine how the son nodes were obtained from the given node. It is a 2-term of success branches and fail branches, which are both lists:

```
empty_branches( branches([], []) ).
```

[20] We use the queue data type implemented in the Quintus package library(queues). This module is implemented using difference lists, and therefore node is an incomplete data structure.

[21] There are more functions implemented in *search_node*, similar to those specified here.

```
successes_of_branches( branches(Successes, _), Successes ).
fails_of_branches( branches(_, Fails), Fails ).
```

A successful branch corresponds to the application of a local fix, a new shift fix, or a retry shift fix. However, a fail branch corresponds to a failure of such an application. We explained how failure can occur in Section 6.1.2. Currently, a branch is equivalent to the applied search fix. The search fixes are defined in *search_fix* (Section 6.5).

6.2.6 father_of_node

Father is the link from the given node back to its father node. It is defined as:

```
void_father( voidfather ).
node_of_father( father(Node, _), Node ).
branch_of_father( father(_, Branch), Branch ).
```

If a node is the root node of a regional search tree, then its father is void as defined by the fact voidfather/1. All other nodes have a compound father with a father node and the branch that produced the given node. If the given node is a root node of an incarnated incarnation, then the branch equals the *incarnate_foreign* that initiated this incarnation.

A search node only has a link to its father and not to its son nodes. Hence, we can only trace a regional search tree bottom-up.

6.2.7 subnodes_of_node

Subnodes were explained in Section 3. They are implemented as an association list in which the key is the region of the subnode and the value is the subnode itself.[22] A subnode is of type node. A node can never be a subnode of one of its own subnodes, and therefore the component subnodes will never become circular.

6.2.8 incarnation_of_node

This component represents the unique identifier of an incarnation. The data type incarnation, which is defined in *search_incarnation* (Section 6.4), corresponds to the

[22] Association lists are implemented in the module assoc.pl which is described in Section 7.2.3. This is not the same module as the one defined in the Quintus package library(assoc).

incarnation concept described in Section 3.2. An incarnation is a data structure serving to group nodes and to limit the search. In fact, a node does not need to know to which incarnation it belongs. Moreover, as we mentioned in Section 3.2, it is possible for some nodes to belong to no incarnation. For such nodes, incarnation_of_node becomes ambiguous. The search never uses this information directly, but uses it only to derive the region of a node via the function provided by *gem*[23]:

% location_of_incid(IncId+, Region−) .

Only *gem* uses this component. However, the search guarantees only that location_of_-incid/2 will transform this component to the unique region to which the node belongs. In a nutshell, for the search, incarnation_of_node/2 acts as region_of_node/2.

6.2.9 sharednodes_of_node

Sharednodes were explained in Section 4.4. A shared node of a node is either a shared subnode or a shared node of one of its subnodes. They are implemented as an association list in which the key is the region of the shared node and the value is the shared node itself. A shared node is of type node. A node cannot belong to its own set of shared nodes, and because of the partial ordering relation among regions, the component shared nodes will never become circular.

6.3 search_userctl

This procedural module allows the user to fine tune the search. Who is the user? The user is the designer of the fixes and constraints in *gem* for a given problem. S/he knows the characteristics of these fixes and constraints for the given problem, and hence s/he is the right person to determine the search control. We saw that the search algorithm is generic, i.e. that many predicates were left undefined. These predicates must be defined in *search_userctl.*

We overview the required predicates, give their specification, and for some, give an example of how they can be defined. However, the definitions are not fixed and are meant to be changed by the designer.

The following predicates are exported, and must be defined by the module:

[23] The need for the unique identifier of an incarnation in *gem* is historical and is of no importance here.

```
:- module(search_userctl, [
        prepare_fix/3,
        satisfied_node/1,
        postpone_node/2,
        prune_node/1,
        put_and_select_node/4,
        select_node/3,
        derive_check_and_fixbugs/5,
        generate_gemconstraints/4 ]).
```

6.3.1 prepare_fix

The function of prepare_fix/3 follows from the definition of search_incarnation/4 (Section 6.1.1). Prepare_fix(Node+, Fix−, NewNode−) investigates Node, and returns a search fix Fix. NewNode is derived from Node and reflects the investigation: prepare_fix/3 may only change the check and expand components of Node to obtain NewNode. These components are explained in detail in Section 6.2. They record information about the constraints and the fixes at that node. Logically, NewNode represents the same node in the regional search tree as Node, and therefore other node components may not be changed. Fix should be one of the types defined in *search_fix* (Section 6.5). The type is used to dispatch the clauses of apply_fix/4, as defined in Section 6.1.2.[24] Prepare_fix/3 should never fail.

As an example we present the high-level definition of prepare_fix/3 currently used in GEMPLAN:

```
:- use_module('search_node.pl', [
        expand_of_node/2,
        replace_expand/3,
        check_of_node/2,
        replace_check/3 ]).

:- use_module('search_fix.pl', [ void_searchfix/1 ]).

prepare_fix( Node, SearchFix, NewNode ):-
        expand_of_node( Node, Expand ),
```

[24] To add a new type of search fix, define this new type in *search_fix*, extend the specification of prepare_fix/3 in the obvious way, and add one clause to the definition of apply_fix/4 in *search_main*.

```
        explore_expand( Node, Expand, SearchFix, NewExpand ), !,
        replace_expand( Node, NewExpand, NewNode ).
prepare_fix( Node, SearchFix, NewNode ):-
        check_of_node( Node, Check ),
        expand_of_node( Node, Expand ),
        explore_check( Node, Check, Expand, NewCheck, NewExpand ),
        replace_expand( Node, NewExpand, TempNode ),
        replace_check( TempNode, NewCheck, NewNode ),
        void_searchfix( SearchFix ).
```

This definition states that we first try to explore the expand component of Node. The definition of explore_expand/4 is not presented here, and it is up to the user to design it. In its current implementation, explore_expand/4 first tries to select a retry shift fix. If there are no retry fixes at Node, it will investigate Node and, if possible, generate either a local, a new shift, or a fail search fix, depending on the result of the call to fix_bugs/5 defined in *gem*. If there is no expand information in Node, explore_expand/4 fails, and the second clause of prepare_fix/3 tries to explore the check component of Node. Again, explore_check/5 should be defined by the user. In the current implementation, explore_check/5 checks the remaining constraints at Node in a circular strategy and produces expand information. The checking of a constraint is performed by a call to check_constraints/4 defined in *gem*. In this case a void search fix is returned.[25] An important requirement of prepare_fix/3 is that it should not fail, or search_incarnation/4 will fail erroneously. This means that the node guards in search_incarnation/4 should catch the nodes for which prepare_fix/3 logically would fail. These node guard predicates are discussed next.

6.3.2 Node guards

The predicates satisfied_node/1, postpone_node/2, and prune_node/1 are called as the first predicate in the body of respectively the first, second and third clauses of search_incarnation/4. The guards were specified in Section 6.1.1. They must be defined by the user in *search_userctl*. We explain how they are currently implemented in GEMPLAN.

satisfied_node/1 It succeeds if all constraints at the given node are satisfied by the plan at that node. This is determined by the check component of node via

[25] It would make sense to explore NewExpand after Check has been explored. Not doing this allows *search_userctl* to stop or postpone the search at the given node and shift attention to another node. This can be interesting because explore_expand/4 can be expensive, e.g. fix_bugs/5.

a function provided by *search_node*, satisfied_check/1. Satisfied_node(Node) implies that the local plan of Node is satisfied, but not necessarily the regional plan of Node.

postpone_node/2 Postpone(Node+, NewNode-) is defined to fail. This means that GEMPLAN currently never postpones a node. In general, however, a NewNode should be returned that will replace Node in the search incarnation for future selection. NewNode should reflect the postpone decision at this node,[26] e.g. postpone_node/2 could check some constraints and return a node that records the bugs in these constraints to be fixed later.

prune_node/1 We prune a node if there are unsatisfied constraints left for which all fixes have been exhausted. In GEMPLAN it will never be possible to find a satisfied node starting from this node, and therefore we discard the corresponding search tree.

Prune_node/1 has an important side purpose of making memory space available for garbage collection. You want to keep the search space small enough so that the set of pages actively used fits into your workstation's physical memory, otherwise the search will perform very poorly. Prune_node/1 forces the search to remove the given node and all of its successors from the current incarnation. The whole subtree with the pruned node as root node becomes inaccessible by the rest of the regional search tree, because there are only links from son nodes to their father, and not from a father node to its child nodes, as we will see in Section 6.2. However, not all of the successor nodes of the pruned node are available for garbage collection, because some of them can still be subnodes and shared nodes of nodes in other regions.

6.3.3 Node selection

The predicates put_and_select_node/4 and select_node/3 determine the flow of control within a search incarnation. Their effect follows from the definition of search_incarnation/4 in Section 6.1.1. Currently, incarnations are implemented as stacks and the search in an incarnation is depth first. Therefore, put_and_select_node/4 simply returns the given node and does not change the incarnation. Select_node/3 calls select_node_inc/3 which returns the top node of the stack incarnation. Select_node_inc/3 is defined in ***search_incarnation*** (Section 6.4).

[26] We come back to this point in Section 7.1.

6.3.4 Constraint and fix generation

The predicates derive_check_and_fixbugs/5 and generate_gemconstraints/4 are called at lower-level definitions of the search which were not explicitly presented in Section 6.1. Derive_check_and_fixbugs/5 is called in local_branch/4 and shift_branch/6 to derive the check component and the expand component of the new son node. Generate_gemconstraints/4 is called in make_incarnations/2 to set up the constraints of the root nodes of the initial incarnations.

derive_check_and_fixbugs(Plan+, Fix+, Node+, Check−, FixBugs−)

Fix is the search fix that is being applied to Node. Plan is the new plan that corresponds to the application of Fix and is derived from the plan of Node. Check is of data type check, which is a component type of node. It describes the state of constraints at a given node. Check describes the initial state of constraints at the son node being derived from Node via Fix. Fixbugs, which is a component of expand, and hence a subcomponent of node, is the initial set of procedures to fix bugs for constraints in Check: Fixbugs will be the fixbugs subcomponent of the son node that is derived from Node via Fix. The current definition in GEMPLAN is too elaborate to present here. However, things will become clearer when we explain the node data type in Section 6.2.

generate_gemconstraints(Region+, Plan+, CheckCs−, SatCs−)

CheckCs and SatCs are both lists of constraints associated with Region. Region is the name of a region. SatCs are the constraints that are satisfied with respect to plan, and CheckCs are constraints that might not be satisfied and therefore need to be checked for Plan. The current definition for GEMPLAN is not presented here.

6.4 search_incarnation

The concept of an incarnation was described in Section 3.2 and used extensively in Section 6.1. This data structure groups a subset of nodes of the same regional search tree. The following procedures are exported by this module [27]:

[27] Actually, two more procedures are exported that are related with the identifier of an incarnation and with the difference between a location and a region. See also footnote 23 on p. 38 and footnote 31 on p. 45.

```
:- module(search_incarnation, [
        expunge_successors/3,
        make_empty_inc/2,
        put_node_inc/3,
        region_of_inc/2,
        select_node_inc/3 ]).
```

It is not difficult to guess their meaning. Currently, an incarnation is implemented as a simple stack, i.e. put_node_inc/3 and select_node_inc/3 act as push and pop respectively. The select_node/3 procedure, which is exported by *search_userctl*, is defined using select_node_inc/3 and implements a simple depth-first search strategy. For this same reason, expunge_successors/3 is currently implemented as

```
expunge_successors( Inc, _Node, Inc ).
```

because we know that Inc has no successor nodes of Node. A more complicated search strategy may be implemented by select_node/3. Therefore, a data structure other than a stack will be needed: Section 7.2.4 gives some hints.

6.5 search_fix

This module implements the data type search fix. We will not present the concrete representation here because it might change as *gem* changes. However, *search_fix* does not depend directly on *gem* and even not on *gem_fix_constraint* (see Figure 11). Search fixes are constructed by *search_userctl* after calls to *gem* using the interface module *gem_fix_constraint*. For example, an *incarnate GEM-fix* defined in *gem_fix_constraint* (see Section 6.6) corresponds to a *foreign incarnate* defined in *search_fix*. Although both have the same concrete representation, *search_userctl* will select the components of the *generate GEM-fix* and then (re)construct the corresponding *foreign incarnate*. These inefficiencies result from abstracting data and are discussed in Section 7.2.1.

Currently, there is only one fix that is created by the search itself and which does not appear as one of the guards in the definition of apply_fix/4 in Section 6.1.2. This fix corresponds to a completion operation as described in Section 4.2. These fixes are created as part of the completion operation performed within shift_branch/6, a predicate called by the last two clauses of apply_fix/4.

6.6 gem_fix_constraint

This module describes the interface between *gem* and *search_userctl*. Essentially, it abstracts the data types of the parameters of the two predicates fix_bugs/5 and check_constraint/4 defined in *gem* and used in *search_userctl*.[28] , [29] Here is the heading of fix_bugs/5, and the definitions of its parameters [30]:

```
% fix_bugs( Selector+, Bugs+, Solutions+, NodeInfo+, GemFix- )

selector_gemfix( fix(Region, ConstraintId, FixId), Region, ConstraintId, FixId ).

gembugs_of_bugs( bugs(GemBugs, _), GemBugs ).
status_of_bugs( bugs(_, Status), Status ).
just_checked_status( justchecked ).
recheck_status( recheck ).

make_gemsolutions( make ).
% list_gemsolutions( Solutions ).

nodeinfo_gemfix( nodeinfo(Node, NewNodeId), Node, NewNodeId ).

fail_gemfix( fail ).
local_gemfix( local(NewPlan, NewBugs, NewSolutions), NewPlan, NewBugs, NewSolutions ).
foreign_gemfix( foreign(GensAndIncs, Plan, Bugs, Solutions),
                GensAndIncs, Plan, NewBugs, Solutions).
incarnate_gemfix( incarnate(ShiftIncId, Father, ShiftPlan, Status),
                  ShiftIncId, Father, ShiftPlan, Status ).
generate_gemfix( generate(ShiftIncId, ShiftPlan, Status),
                 ShiftIncId, ShiftPlan, Status ).

check_foreign_status( check ).
```

[28] *Gem* does not use this module but uses the concrete term representation instead.

[29] There are other predicates of *gem* which are currently called by the search modules. Most are related to the plan representation which is defined in *gem*. They are not abstracted in this interface, because ultimately, a plan should have its own data abstraction, and *gem* should be solely a procedural module. Currently, *gem* is not defined as a Quintus module and belongs to the user module by default. Hence, all the calls from the search modules to predicates defined in *gem* are prefixed by user:.

[30] You will notice that we don't have binary selector/constructor definitions. Using binary definitions is a better way to abstract.

satisfied_foreign_status(satisfied).

The first parameter selects the correct fix_bugs/5 procedure using the region[31] Region, the constraint ConstraintId, and the fix FixId, which should all be instantiated upon calling fix_bugs/5. We wrapped these selectors to make use of Quintus' indexing scheme.[32] Bugs and Solutions provide information for fix_bugs/5. GemBugs of Bugs are the remaining bugs present in the plan of Node with respect to constraint ConstraintId. Status is either *recheck* or *justchecked*. It is justchecked if the constraint has just been checked and the fix is now being applied to the first bug. Solutions is the set of solutions untried by fix_bugs/5 to fix the bugs for the given constraint. If there are no solutions yet present, Solutions should be instantiated to the constant *make*. The members of Solutions represent different alternatives by which the given fix can be executed. Node is the search node at which fix_bugs/5 is called, and NewNodeId is a unique identifier of the possible son node of Node that might eventually result from this call to fix_bugs/5. Finally, GemFix is returned as the result of the fix_bugs/5 procedure, and is of one of the following types:

fail: the fix failed

local(NewPlan, NewBugs, NewSolutions): the fix performed a local change to the plan of Node, resulting in NewPlan. NewBugs represents the remaining bugs in NewPlan. NewSolutions gives the remaining solutions that are left untried and might be an empty list.

foreign(GenAndIncs, NewPlan, NewBugs, NewSolutions): The fix resulted in the sequence of shifts to newly generated regions and to existing regions. This sequence is represented by the list GenAndIncs and should be interpreted left to right. A single shift is represented respectively by:

generate(ShiftInc, ShiftPlan, Status): The fix resulted in a shift to a newly generated region with an initial incarnation identification ShiftInc and an initial local plan ShiftPlan. Status, which is either satisfied or check, represents whether the local ShiftPlan is satisfied or the constraints need to be checked.

[31] In fact, this is not the region but the location_info_name of a region. A location_info_name is defined in *gem*.

[32] Quintus Prolog only indexes on the first parameter.

incarnate(ShiftInc, Father, ShiftPlan, Status): The fix resulted in a shift to a new incarnation of an existing region with identification ShiftInc and an initial local plan ShiftPlan. Father is the father node of the future root node of the new incarnation.

The list GenAndIncs may not contain more than one generate or incarnate to the same region.[33] [34] NewPlan is the update of the plan of Node, such that the ShiftPlan of every generate and incarnate of the GenAndIncs of foreign are included as subplans. NewBugs and NewSolutions have the same meaning as in the case of a local fix.

The following is the heading of check_constraint/4[35] and the definition of its first parameter:

```
% check_constraint( Selector+, Node+, Plan+, Bugs- )

selector_gemconstraint( constraint(Region, ConstraintId), Region, ConstraintId ).
```

Region of Selector is the region[36] in which the constraint ConstraintId is checked. We wrap both parameters with a functor constraint/2 to make use of Quintus's first parameter indexing scheme. Node is the search node at which the constraint is checked. Plan is the plan against which the constraint should be checked. The result of check_constraint/4 is Bugs, a list of bugs. If the constraint is satisfied in the given plan, Bugs is the empty list.

[33] The region can be derived with user:location_of_incid(IncId, Region), which is defined in *gem*.

[34] This statement is not precise. The GenAndIncs component of a new shift search fix may not contain more than one *incarnate_foreign* or *generate_foreign* to the same region. However, this component of a new shift search fix is derived from the GenAndIncs component of a foreign GEM-fix in *search_userctl*.

[35] In fact, the actual definition of check_constraint/4 has one more input parameter, i.e. BadParams. Prior to each call of check_constraint/4, we call another procedure of *gem* that calculates this parameter for a given Region, a given Constraint and a given Plan. BadParams contains information with respect to parameterized constraints. BadParams should probably be calculated within check_constraint/4 so that it would not visible. Therefore, we do not present it here.

[36] See footnote 31

7 Extensions

This section describes the extensions that are currently being made to the system and possible additions to improve the system.

7.1 Satisfaction

In Section 4.3 we saw that the completion operation was necessary to maintain consistency, but that this completion can make previously satisfied nodes unsatisfied. There is also a second reason why a solution node found in an incarnation can be unsatisfied. The user can postpone the search in an incarnation with the predicate postpone_node/2 that he defines in *search_userctl* (6.3). The returned solution node can then be included as the subnode of some higher-level node. If this occurs, it is possible that all of the regional constraints of the higher-level regions are satisfied, but that its subnodes are not.[37]

The definition of *satisfied node*, given in Section 4.3, implies that a node cannot determine locally if it is satisfied. The goal of the search is to find a satisfied and consistent node in the global regional search tree. Whenever we find a solution node in the global incarnation, we know that it is consistent and that we could apply the definition of satisfied node recursively to its subnodes to check whether it is satisfied. This, however, is a bad idea; it would be very expensive because it would entail constantly rechecking all of the constraints of the given problem. This approach also tells us nothing about how to achieve satisfaction from an unsatisfied node.

7.1.1 Recording satisfaction

We will add one component to the node data structure that records whether a node is *satisfied*, *unsatisfied* or *potentially unsatisfied*. From the check component of a node (see Section 6.2.4), we can easily determine if a node satisfies its regional constraints. A node is unsatisfied if one of its regional constraints is unsatisfied or if any of its subnodes are unsatisfied. A node is potentially unsatisfied if it is not unsatisfied and if at least one of its subnodes is potentially unsatisfied. A node becomes potentially

[37] On the other hand, a postponed node that is completed later can become satisfied. In fact, this is our motivation for providing the user with the ability to postpone the search at an unsatisfied node. We will see that this changes the status of an unsatisfied node to a potentially unsatisfied node.

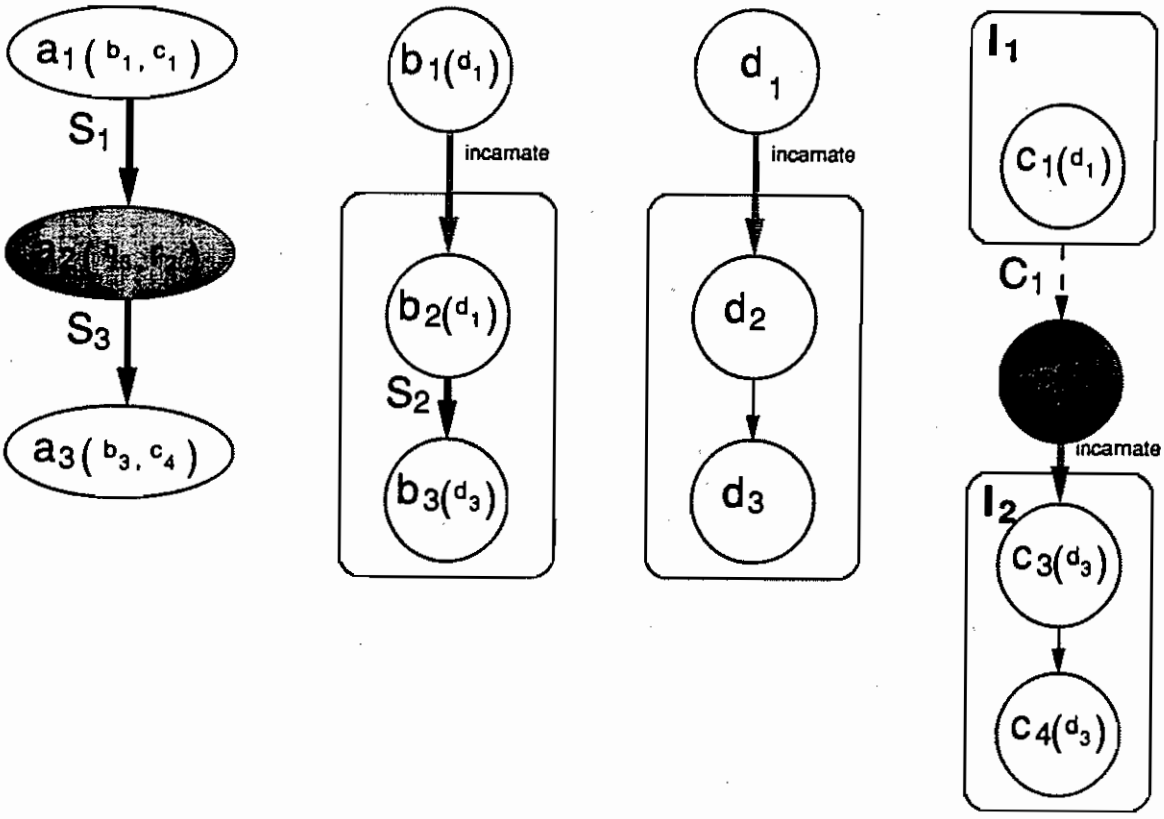

Figure 12: Achieving satisfaction

unsatisfied because of a completion operation. Since the completion operation is recursively applied to subnodes, this node property can be determined for every node involved in the completion process. The nodes completed with a complete fix and all of the nodes including a completed subnode or a new potentially unsatisfied subnode are tagged as potentially unsatisfied.

7.1.2 How to achieve satisfaction

Figure 12 corresponds to the example presented in Figure 5 and Figure 6. After S_1 has been applied, the completion operation will tag c_2 and a_2 as potentially unsatisfied, represented in Figure 12 as shaded nodes. The search control in **A** is at a_2 and decides to make a_2 satisfied. Therefore, it performs a shift fix S_3 to **C**, creating a new incarnation I_2 with root node c_3. In most of the cases, this root node will be a copy of the

potentially unsatisfied node c_2, except for its branches and father. The search in I_2 finds a solution node c_4 which is possibly satisfied. Control goes back to **A**, where S_3 produces a_3 which is no longer potentially unsatisfied. In this case, satisfaction in **A** is achieved with a shift fix to **C**. Achieving satisfaction can be done in multiple ways. In the example, S_3 must not necessarily copy c_2 to create the root node c_3 but can introduce new events to obtain c_3, combining further plan construction and achieving satisfaction within the same shift fix S_3. Also, satisfaction might be achieved at regions other than the global region. The conclusion is that the search mechanism in itself does not achieve satisfaction automatically, but that it calculates and updates the satisfaction properties of the node during the search. To allow for domain dependent control, the search gives full control to *search_userctl* as to when and how to achieve satisfaction. By investigating the nodes, prepare_fix/3 is able to propose the right fixes which eventually must achieve satisfaction for a global solution node.

7.2 Prolog implementation details

Prolog is a wonderful programming language, but it is not a magic tool that solves problems without effort. A Prolog programmer still must use clever techniques and efficient algorithms. The big savings always come from choosing an inherently efficient algorithms, and not from choosing one language over another [5]. Prolog, however, is well suited for developing prototypes by running specifications as programs. This section describes some ways of optimizing our prototype of the localized search algorithm.

7.2.1 Abstract data types

Good Prolog programs are extremely easy to maintain, but large Prolog programs suffer from the same maintenance problems as all other large programs. Abstracting data and procedures is the only way to control this complexity. In our program, the particular term structure of a data type is hidden, selection and construction are done with unary and binary predicates[38], and modification is done with tertiary predicates. Unfortunately, abstracting data in Prolog is inefficient. The following code is actually used to set up a root node in *search_main*.

```
:- use_module('search_node.pl', [
        id_of_node/2,
```

[38]The module *gem_fix_constraint* is an exception.

```
        incarnation_of_node/2,
        plan_of_node/2,
        expand_of_node/2,
        no_fixbugs/1,
        no_retrys/1,
        check_of_node/2,
        no_visiteds/1,
        empty_constraints/1,
        put_front_constraints/3,
        constraints_of_check/2,
        empty_satisfieds/1,
        put_satisfieds/3,
        satisfieds_of_check/2,
        no_checkeds/1,
        no_branches/1,
        subnodes_of_node/2,
        father_of_node/2,
        sharednodes_of_node/2 ]).

:- use_module('search_userctl.pl', [ generate_gemconstraints/4 ]).

%:- use_module('gem', [ location_of_incid/2 ] ).

make_root_node( IncId, NodeId, Plan, Father, SubNodes, SharedNodes, Node ):-
        id_of_node( Node, NodeId ),
        incarnation_of_node( Node, IncId ),
        plan_of_node( Node, Plan ),
        expand_of_node( Node, Expand ),
        no_fixbugs( Expand ),
        no_retrys( Expand ),
        check_of_node( Node, Check ),
        no_visiteds( Check ),
        user:location_of_incid( IncId, Region ),
        generate_gemconstraints( Region, Plan, GemConstraints, GemSatisfieds ),
        empty_constraints( Constraints ),
        put_front_constraints( GemConstraints, Constraints, NewConstraints ),
        constraints_of_check( Check, NewConstraints ),
        empty_satisfieds( Satisfieds ),
        put_satisfieds( GemSatisfieds, Satisfieds, NewSatisfieds ),
```

```
        satisfieds_of_check( Check, NewSatisfieds ),
        no_checkeds( Check ),
        no_branches( Node ),
        subnodes_of_node( Node, SubNodes ),
        father_of_node( Node, Father ),
        sharednodes_of_node( Node, SharedNodes ) .
```

Without the node data abstraction this same predicate would have the following definition.

```
%:- use_module('gem', [ location_of_incid/2 ] ).

:- use_module('search_userctl.pl', [ generate_gemconstraints/4 ]).

make_root_node( IncId, NodeId, Plan, Father, SubNodes, SharedNodes,
                node( NodeId,
                      Plan,
                      expand( [], [] ),
                      check( Satisfieds, [], Constraints, [] ),
                      branches( [], [] ),
                      Father,
                      SubNodes,
                      IncId,
                      SharedNodes ) ):-
        user:location_of_incid( IncId, Region ),
        generate_gemconstraints( Region, Plan, GemConstraints, GemSatisfieds ).
```

The second definition may run ten times faster. Certainly, this is worth of concern. However, the clarity and maintainability advantages of the first definition are compelling. We have no simple solution to this dilemma.[39]

7.2.2 Transitive closures

The regional structure of the problem domain is described with the partof/2 facts. We defined t_partof/2 as its transitive closure.[40] The definition that is used by *gem* is:

[39] Unfolding is only a partial solution.

[40] All of these region structure definitions are in *gem*. T_partof does not actually exist, but similar transitive closure operations are defined.

```
t_partof( R1, R2 ):- partof( R1, R2 ).
t_partof( R1, R2 ):- partof( R1, R3 ), t_partof( R3, R2 ).
```

This is superior as a specification, but it is also amazingly inefficient. There are two possible solutions. First, the Quintus package library(graphs) implements Warshall's algorithm which calculates the transitive closure of a graph in $\bigcirc(N^3)$. Second, we can pre-calculate t_partof and assert it as facts because the regional structure is static. Transitively closed predicates are used extensively during completion of a node and during updating of the shared nodes. An efficient implementation such as this will speed the search.

7.2.3 Association lists

Association lists are implemented in the module *assoc.pl*. Shared nodes, subnodes and some other temporary structures constructed during completion use association lists. Association lists are represented as lists of (Key–Val) pairs. This is advantageous because potentially many library predicates are applicable to them. We didn't choose the Quintus package library(assoc) that represents association lists using binary trees because our application of association lists is dynamic; the tree would quickly become imbalanced, and the overhead of keeping it balanced would be too great. The best implementation would be to order association lists on their keys, since frequently used operations, like merging lists and selecting values with given keys, could be implemented more efficiently. Association lists can be ordered with the built-in predicate keysort/2.

7.2.4 Incarnation revisited

Section 6.4 showed that a search incarnation is currently implemented as a stack. A stack is the best choice if the search strategy in an incarnation is depth first. If we want to allow the user to define other strategies via the predicates select_node/3 and put_and_select_node/4,[41] then a more complex representation must be developed, possibly one with an indexing scheme on the plan and the identification component of the nodes in the incarnation. Moreover, for almost any strategy other than depth-first, the definition of expunge_successors/3 becomes nontrivial. In Section 6.2, we saw that a regional search tree only has bottom-up links, and hence it is impossible to trace

[41] It is very likely that arguments will have to be added to these predicates.

down the successor nodes from a given node. Hence, the incarnation must record these links so expunge_successors/3 can be implemented efficiently.

8 Conclusion

This report describes a generic localized search algorithm that makes constraint localization possible. The algorithm was explained in detail by including the top-level Prolog code. The search is generic and allows a multitude of search strategies determined by the particular implementation of a user-defined module. The current prototype uses Quintus' module facility and abstracts data types to make maintaining and future modifications tractable. Finally, some clues were given to improve the efficiency of the current system.

References

[1] Lansky, A.L. "Localized Representation and Planning," The 1989 Stanford Spring Symposium–Workshop on Planning and Search (March 1989).

[2] Lansky, A.L. "Localized Event-Based Reasoning for Multiagent Domains," *Computational Intelligence Journal, Special Issue on Planning*, Volume 4, Number 4 (November 1988). Also appeared as Technical Note 423, Artificial Intelligence Center, SRI International, Menlo Park, California (1988).

[3] Lansky, A.L. "A Representation of Parallel Activity Based on Events, Structure, and Causality," in *Reasoning About Actions and Plans, Proceedings of the 1986 Workshop at Timberline, Oregon*, M. Georgeff and A. Lansky (editors), Morgan Kaufmann Publishers, Los Altos, California, pp. 123-160 (1987). Also appeared as Technical Note 401, Artificial Intelligence Center, SRI International, Menlo Park, California (1986).

[4] Lansky, A.L. and D.S. Fogelsong. "Localized Representation and Planning Methods for Parallel Domains," in *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, Seattle, Washington, pp. 240-245 (1987).

[5] O'Keefe, Richard A. "Prolog Compared with Lisp," *SIGPLAN Notices*, Volume 18, Number 5, May 1983, pp. 46–56.