The AMOS Project

IST-2001-34717

Selection Heuristics for Matching

Deliverable D7

Responsible person: Manuel Carro, Francisco Bueno, Daniel Cabeza, Carlo Daffara, Technical University of Madrid and Conecta s.r.L. ({`mcarro,bueno,dcabeza`}`@fi.upm.es`, `cdaffara@conecta.it`)

Date of current release:	May 2003
Type of deliverable:	Report

Abstract

In this paper we devise and describe the selection heuristics implemented in the matching engine of AMOS. We divide the search heuristics into two types, depending on their aims: those which try to generate more assemblies, by using safe or unsafe rules, and those which try to present to the user the more promising assemblies, thereby effectively reducing the number of choices the user has to make.

Contents

1 Introduction

The AMOS Project aims at developing a method and a tool to help to categorize and search among software assets for which there is source code available. These are categorized based on an ontology [Daf02] which makes it possible to classify software packages in a uniform way using *tags* and representing entities with a well-defined meaning.

The process of entering descriptions of new packages admittedly needs more work than in other approaches. In the project we also strive to minimize this effort by providing an interface which makes it easier to enter new data for most cases. On the other hand, identifiers with unique meanings make it possible to implement searches which are able to retrieve items with a 100% recall precision. Comparisons of different approaches for software retrieval [MMM95] have highlighted the advantages of this kind of search. We expect that the benefits in the search, much more frequent than information additions, will outweigh the additional effort in insertion.

Highlighting the differences of the Amos approach with other software repositories, such as SourceForge or FreshMeat, is interesting. Amos does **not** aim at being a software repository, but rather just at holding descriptions of software, instead of the software itself. It aims at taking advantage of accurate descriptions, which can be richer and, in practice, more flexible, than the tree-like classification most software repositories provide. As we will see, the kind of searches Amos offers is also more powerful, while accuracy is not sacrificed unnecessarily.

It is also different from the RPM, Apt, Yum, or Portage repositories in that no binaries or sources themselves are necessarily part of the database, and the descriptions are based on a set of predefined dictionary terms which after an initial fill-in are expected to change very slowly over time. Package descriptions do not refer to other package identifiers / names, but to the capabilities packages offer or need. Amos is certainly not targeted to the average computer user who wants to download software, install, and use it. It is addressed to the software developer who wants to retrieve, and very possibly modify, the source code of a series of packages (i.e., an *assembly*) which satisfy a set of requirements. To this end, the search engine should be able to chain search steps to determine one or more assemblies of inter-dependent packages which satisfy as completely as possible a set of initial, user-stated requirements.

Since several packages might provide the same capability (or set of capabilities), several assemblies are possible which could solve an initial query. In order to better suit developer's needs, different heuristics will be made available. Also, and since packages can ultimately be changed by the developer, approximate searches are supported by using *generalizations* of terms stated as such in the dictionary: a term can be registered to be a generalization of a more precise one, and this knowledge can be used to expand and guide the search to packages featuring alternative (but related) descriptions.

In this report we will focus on searches which use dictionary items to describe a package, with the proviso that sorting heuristics can also be applied *a posteriori* to other fields of the package description. We will also pay attention to (and describe a number of them) heuristics designed to prefer to some assemblies (more advantageous for a user) over others.

The rest of the report will proceed as follows: Section 2 will present a simple model for the representation of packages (Section 2.1), the search procedure, and the use of generalization rules (Section 2.2), we will discuss alternative uses of these rules, and we will also sketch how situations in which there are too many solutions can be handled (Section 2.5). Section 3 will refine, from a user-directed point of view, the alternatives put forward in the previous sections in order to devise a user-oriented interface.

2 A Logical View of Packages

In order to capture the idea and usage of package descriptions and generalizations, and to showcase the kind of searches and heuristics Amos offers, we will resort to a representation using first-order logic. This notation is well known and easy to understand, while at the same time adequate for our purposes.

The structure of a package description in the ontology has several fields, including, e.g., author, WWW site where the software proper is stored, number of lines of code, etc. Not all of them are actually useful to perform searches (because they do not have a unique associated meaning): for example, deciding whether the WWW address associated to the package, the name of the (main) author, and other similar fields are really meaningful is best left to the user. They can, however, be used to sort the results according to some heuristic, or be used as an auxiliary filter by using, for example, string matching.

2.1 Package Requirements and Capabilities

A package P is minimally described with a set of terms, drawn from a predefined dictionary, which state:

- What is provided by the package (i.e., what capabilities other packages can expect from P), and
- What is needed by this package (i.e., what capabilities, usually provided by other packages, are needed for P to work properly and as a complete software package).

Examples of terms used to denote capabilities (i.e., dictionary terms) are: `compiler`, `c_compiler`, `java_runtime_environment`, `c++_compiler`, `java_class`, `java_compiler`, `std_c_libraries`, `source_ada_program`, etc. A package named `ada_to_java_compiler` can be described, for example, as having the requirement of a `source_ada_program` and providing a `java_class`.

A package p requiring the set of terms req and providing the set of terms $prov$ is represented by the facts $Requires(p, req)$ and $Provides(p, prov)$. We also assume that $prov \cap req = \emptyset$, i.e., a package should not need a capability provided by itself.[1]

A search starts by stating which capabilities are required in a project, and ends up with a set of software packages satisfying as many requirements as possible. Thus, we will represent a state in the search by a pair $(pack, r)$ where $pack$ is a set of packages and r a set of dictionary terms. In a given state, $pack$ is the current assembly of packages, and r the required capabilities that are not yet fulfilled by $pack$.

The initial state of a search for an assembly that provides a set req of capabilities is thus $(\emptyset, req)$. The ideal final state should be $(pack, \emptyset)$, where assembly $pack$ satisfies all initial requirements. However, some requirements may be left unsatisfied because, e.g., no package providing them is available in the queried database.

Every step in the search selects a package which, hopefully, will reduce the set of required capabilities. Such a selection augments the packages in the current assembly with one more package, adds the requirements of the added package to the non-fulfilled capabilities, and subtracts from them the capabilities provided by the added package. We express this by the following search rule:

$$\frac{(pack, r)\ ,\ Provides(p, prov)\ \wedge\ Requires(p, req) \wedge r \cap prov \neq \emptyset}{(\ pack \bigcup \{p\},\ (r \bigcup req) \setminus prov\ )} \tag{1}$$

[1]This is not strictly necessary, since such redundancies can be removed at runtime by the search algorithm.

A search consists in a repeated application of this rule. The rule can be applied as many times as necessary, usually until there is no package p which can provide any of the pending requirements (i.e., $r \cap prov \neq \emptyset$ does not hold, so that the rule cannot be applied). Note that $r \cap prov \neq \emptyset$ guarantees that the selected package satisfies some requirements, but this does not necessarily mean that the set of requirements is reduced (the package added can have its own new requirements). Two remarks can be made:

- The selection of the package to be chosen at every search step is external to the rule itself. Several strategies can be adopted to guide the search: for example, giving preference to packages which satisfy more requirements (i.e., have the biggest intersection $r \cap prov$), or which effectively reduce the set of pending requirements (i.e., $req \subseteq r$), or which augment the least the set of pending requirements (i.e., $(r \cup req) \setminus prov$ is the smallest), There is plenty of room for heuristics.

- In order to be as informative and efficient as possible, more data has to be carried during the search. For example, the set of capabilities which are actually provided by the accumulated packages in the current state are not explicit in rule (1). Not using this set might cause the same package to be used twice, or use packages not strictly needed: they can be added to fulfill a requirement of some other package that is already satisfied by another package in the assembly. While adding packages twice does not change the final result of the search, it has to be avoided in a real implementation for efficiency reasons. Also, adding "spurious" packages is not a good idea, either. Additionally, as we will see, the information on the capabilities provided by an assembly can be interesting to the user of the tool. Making it part of the search algorithm is straightforward, and we will assume that it is available when needed.

The algorithm presented in [Car03], and shown here in Figure 2, performs this search exhaustively in a backward-chaining fashion. The current implementation in the tool uses a classical technique (inverse indexes) to speed up even more the execution.

2.2 Dictionary Terms and their Generalizations

A term used to describe packages is in principle only a syntactic entity, and it is handled as such by the matching engine. Terms are collected in a dictionary which provides also a (short) explanation of their intended meaning in human language. Users and package implementors use these definitions to agree on the meaning of the dictionary terms. The matching engine makes no use of the meaning of terms; it simply matches terms syntactically.

The dictionary contains also *generalizations* of terms: it states when a term is an instance of a more general one. For example, a `c_compiler` is an instance of the more general term `compiler`. Any package providing a `c_compiler` provides also a `compiler` (but not necessarily the other way around); a package requiring just a `compiler` should do with a `c_compiler` (but, again, probably not in the opposite direction).

There are other cases where this relationship is clearer (and more feasible in practice): for example, different filters for analog signals are known to be instances of more general filters. Some packages might offer the full range of filters, while some other packages might offer only a subset of them. In this case it makes sense to use a more general term to describe the capabilities of the package offering general filters, so that the use of generalizations is thus easy to justify.

A generalization has the following semantics: if term t is generalized by term t_g (which we will write $t \Rightarrow t_g$), then

$$\forall p : Provides(p, prov) \wedge t \in prov \rightarrow Provides(p, (prov \backslash \{t\}) \cup \{t_g\}) \qquad (2)$$

Note that we "infer" a new fact about term p where the generalized term t does not appear. This is so in order to separate and, if necessary, to mark the knowledge which derives from such an inference from the knowledge which is explicit in the package database. This can be useful in order to, e.g., trace which inferences have been done during a search. The intended meaning is that t_g is an abstraction of t: for example, the idea of `compiler` (meaning *some* compiler, but not *every* compiler), is a generalization of, for example, `c_compiler`.

Figure 1 shows a depiction of generalization relations. Generalizing a term boils down to moving from a node to some parent in the lattice. Moving in the opposite direction (from a node to one of their children) finds out a more concrete term for a more general one. Within a search procedure, generalization relations can be used in *safe* and *unsafe* ways, described below.

2.3 Applying generalizations safely

According to definition (2), if $t \Rightarrow t_g$ then every package which provides a capability t provides also the more general capability t_g, i.e., a capability of which t is a particular

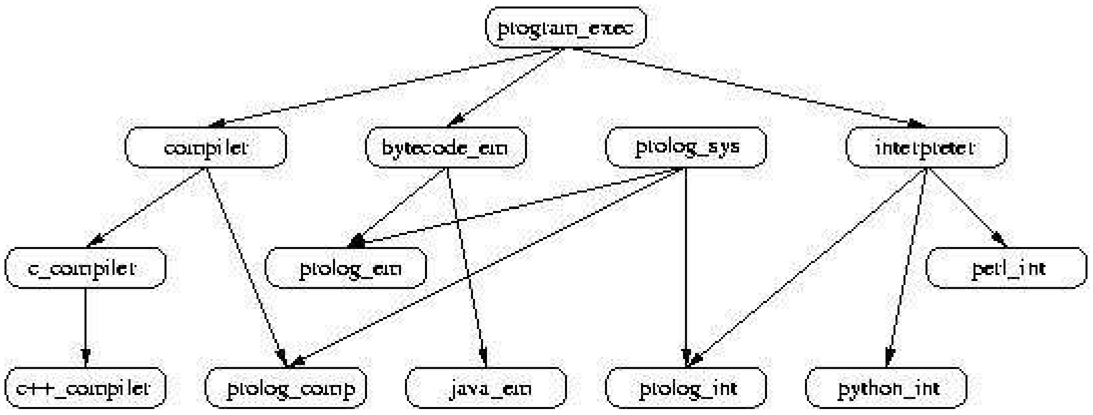


Figure 1: A lattice depicting generalization relations

case. Taking this into account, we can write a generalization rule, which will interact with the search rule, as follows:

$$\frac{t \ \Rightarrow \ t_g \ , \ Provides(p, prov) \wedge t \in prov}{Provides(p, (prov \setminus \{t\}) \cup \{t_g\})} \tag{3}$$

Intuitively, new facts about what a package provides can be deduced by using the generalization rule together with information already existing about a package. These new facts can be used by the search rule in the same way as those initially in the database. They can also be used in new applications of the generalization rule itself. For example, since a `c_compiler` is more concrete than a `compiler`, package providing a `c_compiler` certainly provides also a `compiler`.

Symmetrically, generalizations also interact with the required capabilities as follows:

$$\frac{t \ \Rightarrow \ t_g \ , \ Requires(p, req) \wedge t_g \in req}{Requires(p, (req \setminus \{t_g\}) \cup \{t\})} \tag{4}$$

I.e., a package which requires a `compiler` should be happy with **any** compiler, or a package requiring just a `text_editor` should be happy with anything that qualifies as `text_editor`, for example, an `editor_syntax_coloring`. The new $Requires$ facts can again be used to search for assemblies.

However, we want to note that it is redundant to apply generalizations both to the $Provides$ and $Requires$ facts of packages, and applying only on one side is enough. Without a formal proof, it can be seen that if a generalization rule is needed to help the search progress, it can be applied either to the provided capabilities (when looking for a candidate package) or to the required capabilities (when adding them to the set of pending requirements), because the generalization rule stands “in between” the $Requires$ and $Provides$ chain, the LHS of the generalization being tied to the provided capabilities and the RHS of the generalization being associated to the required capabilities.

With rule (4), if we add to an assembly a new package that requires t_g and we have $t \ \Rightarrow \ t_g$, the new status will have requirement t, instead. This requirement might be later fulfilled by another package p. The effect would have been the same if we had added requirement t_g and used rule (3) on the $Provides$ fact for p. Thus, the results are the same whether we *fire* the use of generalizations to effectively generalize on the provided capabilities, or to concretize on the required capabilities. This simplifies the implementation of the search procedure, which can only take into account one of the rules.

2.4 Unsafe generalizations

The generalization rules can be also be applied in the opposite direction:

$$\frac{t \ \Rightarrow \ t_g \ , \ Provides(p, prov) \wedge t_g \in prov}{Provides(p, (prov \setminus \{t_g\}) \cup \{t\})} \tag{5}$$

$$\frac{t \ \Rightarrow \ t_g \ , \ Requires(p, req) \wedge t \in req}{Requires(p, (req \setminus \{t\}) \cup \{t_g\})} \tag{6}$$

These rules are logically unsound. To explain this, let us concentrate on the first one, since it turns out that the second one is redundant for the search, much the same as what happens with rules (4) and (3).

Rule (5) expresses that whenever a feature is provided, a more concrete *subfeature* is also provided, which is clearly contrary to the intuition we gave before. This is akin to assuming that definition (2) also holds in the opposite direction. I.e., a package providing a `compiler` would provide as well a `c_compiler`, which is not necessarily so. The reason to allow the application of this rule stems from the possibility of changing the source code of some package to adapt it to purposes different from those it was initially devised for. Using it in conjunction with the safe generalization rule provides a way to *jump* to sibling nodes (and then to cousins, and so on), which are expected to be related in a decreasing degree. A limited[2] number of steps permits then using

[2]It is reasonable to limit the number of unsafe generalizations, as we would otherwise have a complete traversal of the package database.

more terms to look for package matches in the same way as those explicitly listed in the package descriptions.

For example, the class of sorting algorithms which remove duplicates (say, which provide `sort_no_dup`) and that of sorting algorithms which do not remove duplicates (`sort_dup`) are instances of `sort` algorithms. Should the user be looking for a non-removing sort algorithm, if all packages in a given database which provide the ability to sort are of the *removing* type, unsafe generalizations would make it possible for the search procedure to use the `sorting` (i.e., any sorting algorithm) requirement from the more particular `sorting_dup` (this is an unsafe step) and then use the requirement of a `sorting_no_dup` (this step is safe). The code for the retrieved package can be later modified by the programmer in order not to remove duplicates.

Input: R, a set of requirements
Output: (P, R), sets of needed packages and unfulfilled requirements
Algorithm:

```
F := $\emptyset$            -- What has been fulfilled so far
P := $\emptyset$            -- Packages used so far
do                 -- Invariant:  $R \cap F = \emptyset$
     select $A$ s.t. $Provides(A, q) \wedge Requires(A, p) \wedge q \cap R \neq \emptyset$
     F := F $\cup$ q
     R := (R - q) $\cup$ (p - F)
     P := P $\cup$ {A}
until <there is no $A$ s.t. $Provides(A, q) \wedge q \cap R \neq \emptyset$>
return (P, R)
```

Figure 2: Schematic nondeterministic search algorithm

2.5 Searching for Packages

The search algorithm in Figure 2 (from [Car03]) starts with an initial set of requirements and looks for packages which offer some of them in order to progressively reduce that set until no more reduction steps can be performed. The algorithm is very general, and can be frobbed to suit different needs for the search. We will present here some opportunities for improvement, and discuss them more thoroughly and from a user point of view in Section 3.

Finishing the Search The deduction rule does not state any search strategy nor decide when the search has finished, and there is complete freedom to decide which package is to be selected in each search step, and when the search has finished. The non-deterministic search algorithm in Figure 2 stops when the number of packages cannot be reduced. However, other stop criteria could be used.

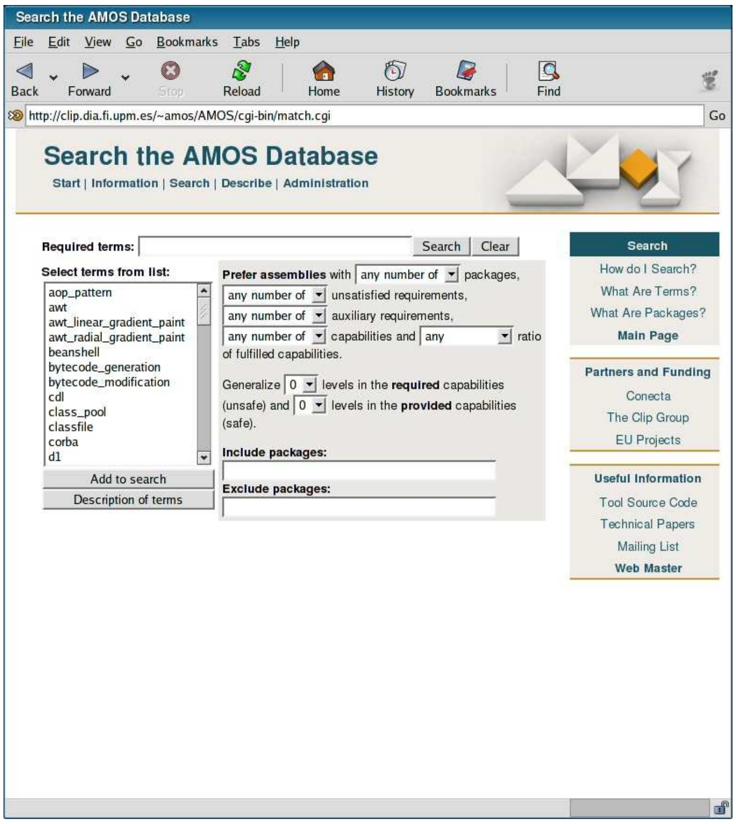
For example, one might choose to finish (a branch of) the search when no more initial requirements can be satisfied, or when the number of unsatisfied capabilities is minimal. In the first case, the search will stop if new packages will only satisfy requirements which were not in the initial requirement set. In the second case, it will stop if adding new packages will only increase the number of pending requirements (even though there might still be some initial requirements pending which could have been satisfied).

In general, stop criteria involve minimizing some function on the state of the search. However, since the final objective is to fulfill a set of initial requirements with a set of packages, it makes sense to make the search try to (eagerly) complete the requirements as much as possible with the packages present in the database, i.e., progress until the number of pending requirements which can be satisfied is, hopefully, zero.

Enlarging the Search Space The application of generalization rules can be dealt with by implementing the `select` keyword conveniently, so that set of package descriptions the selection takes into account is enlarged by generalization.

Guiding the Search and Selecting Assemblies Plugging in search heuristics based on selection rules which use local information to guide the search to a satisfactory assembly faster is also possible by adapting the implementation of `select`. On the other hand, filters which decide which assemblies are more promising typically need information about the rest of the assemblies at every moment.

Influencing the Packages in the Solutions Sometimes an experienced user may want to explicitly ban some packages from a final assembly (because e.g. they have well-known problems or compatibility issues), or force them to be included in the final assembly. It is possible to do that quite straightforwardly by using the algorithm in Figure 2, and simply initializing adequately the search state variables and instructing `select` not to take the banned packages into account.


Search the AMOS Database
File Edit View Go Bookmarks Tabs Help
Back Forward Stop Reload Home History Bookmarks Find
Go

http://clip.dia.fi.upm.es/~amos/AMOS/cgi-bin/match.cgi

Search the AMOS Database

Start | Information | Search | Describe | Administration

Required terms: [Search] [Clear]

Select terms from list:

aop_pattern
awt
awt_linear_gradient_paint
awt_radial_gradient_paint
beanshell
bytecode_generation
bytecode_modification
cdl
class_pool
classfile
corba
d1

[Add to search] [Description of terms]

Prefer assemblies with [any number of] packages,
[any number of] unsatisfied requirements,
[any number of] auxiliary requirements,
[any number of] capabilities and [any] ratio of fulfilled capabilities.

Generalize [0] levels in the **required** capabilities (unsafe) and [0] levels in the **provided** capabilities (safe).

Include packages:

Exclude packages:

Search
How do I Search?
What Are Terms?
What Are Packages?
Main Page

Partners and Funding
Conecta
The Clip Group
EU Projects

Useful Information
Tool Source Code
Technical Papers
Mailing List
Web Master

Figure 3: Search Page

3 Search Parameters

The interface presented to the user has to give an intuitive access to the internal search capabilities, and permit the user to expand the search space in order to find more assemblies if needed. It must also makes it possible to sort the results according to some heuristics without removing solutions. Figure 3 depicts the search page currently implemented. In this section we will describe:

- How the user can specify that the search should use the generalizations in the dictionary in order to generate more (potentially incorrect) results (Section 3.2), and
- How it is possible to have a basic control on which packages are to be forced to appear / not to appear in a final assembly (Section 3.3),
- How the order in which results are shown can be changed by setting some parameters related to different aspects of the quality of an assembly (Sections 3.4 and 3.5). Additional quality measures can be defined, and our approach is easy to update to take them into

We will focus here on designing and discussing search and sort heuristics and parameters, and on justifying their usefulness from the point of view of the final user, rather than on describing their implementations.

3.1 Basic Search

Searching consists, in its most basic form, on giving the tool a set of dictionary terms, which describe desired capabilities, and let the search algorithm to return a set of packages which provide these capabilities. The interface allows to refine this search incrementally.

3.2 Expanding the Search

As already mentioned, term generalization can be used to take more packages into account. Deciding whether to apply (or not) safe and unsafe generalizations must be easily available and clearly distinguished in the user interface. However, an unbound number of generalizations can end up making the search procedure to traverse all of the packages.[3] Being able to limit how many times the generalization rules are used in each search is then a must. We propose the two following control parameters:

Number of safe generalizations This is the number of safe generalizations (Section 2.3) which can be performed when searching for any assembly. If zero, then no safe generalizations will be attempted.

Number of unsafe generalizations This is the number of unsafe generalizations (Section 2.4) which can be performed when searching for any assembly. If zero, then no unsafe generalizations will be attempted.

When both parameters are zero no generalization rule is used, and only the package database is consulted to build assemblies. The larger the number of generalizations, the larger the set of assemblies which can be built.

Safe and unsafe generalizations can be active together, and the result of applying them is to walk up and down the dictionary of terms. The amount of "walking" depends on the "tightness" of the web of term generalizations. As an example, a request for a `c++_compiler` can be satisfied in the end by a package providing a `c_compiler`, by chaining the two generalizations put forward in the previous paragraph. Even this chaining can be interesting if, for example, the code to be compiled does not really depend on object-oriented C++ characteristics and porting it to plain C is relatively easy.

3.3 Controlling Packages in the Final Assembly

Generalizations of either case can produce too many assemblies, probably not all of them of interest (recall that unsafe generalizations can produce logically unsound results). The number of solutions can be tuned by letting the user have some control about which packages are to be used during the search by:

- Specifying which packages are to be included in a final assembly. If these packages fulfill a subset of the initial requirements, then no alternatives to meet these requirements are needed. Note that since we let the search process to finish with unsatisfied capabilities, adding more search terms to the initial query (a

[3] This depends on the particular shape of the generalization tree (or, in general, web), which can connect or not the whole universe of terms.

technique used with Google and other web search tools) will not produce more concrete results.

- Putting a ban on packages also helps to retrieve better assemblies, since software pieces already known (e.g., by personal experience) to be defective can be removed right from the beginning.

We believe that these two possibilities are interesting and therefore they should be included among the options users have available.

Packages not to be taken into account The names of packages which must not be part of any final assembly.

Packages forced to appear in an assembly The names of packages which the user wants to appear in all final assemblies (and which will be taken into account in any intermediate search step).

These two parameters might however fail to reduce sufficiently the number of assemblies and, besides, they need some experience and knowledge about the package database and about the behavior of packages on it. Therefore it is necessary to design an intuitive method to make it easier for the user to select among a large number of solutions, many of which may not be interesting. Since we do not want to remove solutions which have been generated according to user preferences, we have opted for a way to express a ranking on the solutions. We will devote Sections 3.4 and 3.5 to this issue.

3.4 Preference Heuristics

A set of preferences which makes it possible to choose some assembly over the rest can be stated. This would help to reduce the number of assemblies a user has to deal with. We will define a series of quality measures applicable to each assembly in order to rank it in relation with the rest of the assemblies. The ranking will be used to sort the assemblies before presenting them to the user, so that those with a higher score are seen before. In general, this can only be done when all the assemblies for a given initial search are known, and therefore the whole search space has to be exhausted anyway.

Note that these quality measures do not work at the level of the source code: on one hand they address a quite abstract measure (appropriateness of a set of open source packages), which is quite independent of the quality of every source code package, and, on the other hand, metrics addressed to evaluate source code cannot be used here, because in general the Amos database does not have the source code stored. Our metrics work on the following data, which is currently available [Car03] after the solution for each assembly has finished:

Initial search terms These are the terms the user wanted to search for, and they are of course known when the search finishes.

The packages in the final assembly These constitute the core of the information returned to the user.

The unsatisfied requirements These are the requirements which the user wanted to look for at the beginning of the search, and which the current package assembly could not satisfy, plus the requirements needed by the intermediate search steps (i.e., by the packages in the assembly) which the user initially did not require, and which were left unsatisfied.

The satisfied requirements These are the requirements which were needed during the search, either because they were initially stated by the user, or because they were needed by some of the packages in the assembly, and have not bee satisfied. This set is currently not shown to the user in the GUI, but it can be used to guide the search as well.

More information can be gathered during the search; however this would very probably increase the search time, and it was decided not to add at the moment more items to the information carried around during the search in order to have a fast tool. Additionally, the data items above are what the search algorithm needs to avoid redundancies in the search and to decide when to stop. We argue that meaningful data regarding high-level manageability of the assemblies can be generated out of that data.

3.4.1 Size of Assemblies

An approximate measure of the complexity of an assembly is the number of packages in it. For assemblies which satisfy the initial requirements to a similar degree, the number of packages may give a rough indication of how manageable will the assembly be. Assemblies with few packages can be advantageous because they should need less glue code. On the other hand, an assembly consisting of a large number of packages may be easier to adapt to some needs and it may offer additional capabilities, not initially needed, but which may be interesting when planning for future developments.

Number of packages in the final assembly The number of packages in a final assembly can be rated positively (the more packages the better) or negatively (the fewer packages the better).

3.4.2 Unsatisfied requirements

Because of the way the search algorithm works, it is guaranteed that every requirement which remains unsatisfied when a solution is found is not provided by any package in the database. In the absence of a measure of how important each requirement is, the number of unsatisfied requirements roughly indicates how well an assembly satisfies requirements.

Number of unsatisfied requirements The number of requirements which are left unsatisfied at the end of the search.

3.4.3 Redundant capabilities

Redundant capabilities are those finally provided but which were not initially requested. These indicate that the assembly has probably more parts / components than strictly needed. In an extreme case, it would create *bloated* assemblies. If several assemblies have the same number of packages or unsatisfied requirements, the one having more redundant capabilities has probably more (unnecessary) code or more unnecessary internal complexity.

Number of redundant capabilities The number of capabilities which were satisfied during the search, but which were not requested by the user.

3.4.4 Provided capabilities

The total number of provided capabilities (either initially requested or not) is also a measure of the adequacy of the assembly.

Number of provided capabilities The total number of capabilities provided by the final assembly.

3.4.5 Ratio of unmatched requirements

The previous measures are absolute: they compare directly assemblies which can be of different size, and these comparisons can be unfair or, at least, misleading. Preference metrics which are relative to some measure of the size of the assembly would help to generate rankings which do not depend on this size. For example, comparing the number of pending requirements against those that have been satisfied is a way to reduce the relevance of the size of the packages while at the same time deciding how self-contained is the assembly.

Ratio pending / satisfied requirements It returns the value of dividing the number of pending requirements by the number of requirements satisfied in a final solution. The lower this ratio, the better the expected quality of the assembly.

3.5 Combining Preferences

The quality parameters stated above can be at odds with each another, and a means to combining them in a multi-purpose ranking must be defined. A linear combination of the scores of the previous parameters, where the coefficients can be provided by the user, can be used to define an optimization to achieve. In order to give a more friendly interface, we have decided to present the user with symbolic names which are then translated into numeric weights (typically 3 to 5). The final ranking will be generated using the formula

$$r = \sum_{1=1}^{5} p_i w_i$$

where r is the value according to which some assembly is ranked, p_i is the value of i-th parameter and w_i is the weight assigned to it by the user.

4 Conclusions

We have introduced package descriptions and dictionary terms and their generalizations, and shown how they can be conceptualized at the same level and therefore handled uniformly by the search process. We have therefore related this representation model with a skeleton of the search algorithm and shown how this algorithm can be changed by adjusting the behavior of just one line.

We have also seen how generalizations can be used to expand the search using two very different ideas: safe generalizations, which lead to logically correct results (according to the logical reading of term generalizations), and unsafe generalizations, which can lead to incorrect results (again, according to the same reading).

Finally, we have argued that it is possible that too many results are returned, and a way to help the user to handle this amount of information is needed. We have found it reasonable to provide a series of parameters which implement a measure of the quality of the assemblies to sort the different results. Those with a higher score, and which are probably more promising, are presented before to the user.

References

[Car03] M. Carro. The Amos Project: The Matching Engine Design. Technical Report CLIP2/2003.1, Technical University of Madrid, School of Computer Science, UPM, February 2003.

[Daf02] Carlo Daffara. An ontology for open source code. Technical report, Conecta s.r.l., 2002. Deliverable D2 of the AMOS Project.

[MMM95] H. Mili, F. Mili, and A. Mili. Reusing Software: Issues and research directions. *IEEE Transactions on Software Engineering*, 1995.