The Amos Project: an Approach to Reusing Open Source Code

Manuel Carro*
mcarro@fi.upm.es

Computer Science School
Technical University of Madrid
Boadilla del Monte, 28660 Madrid, Spain

Abstract. Building reliable software products based on components whose properties are well established and understood is one of the key goals of component-based software development. There are some cases in which this approach has to face practical problems: for example, when development is based on code not formally developed, and development in the presence of thousands of components. We describe the basis and current work in Amos, an IST-funded project that aims at facilitating the search and selection process of source code assets in order to ease its reuse. We show how, although the description of source code *packages* is generated using a non-formal method, the search and assembly process is done at a formal level, assuming the package characterization is a faithful description. We argue that lessons learned from this approach will be of use to learn about the characterization of software code not formally developed, and about the search for the right component amongst thousands of other components.

1 Introduction and Motivation

Motivations for this paper:

- not components, in a strict sense
- however, something in common:
  - once packages are characterized, then the reasoning is *formal*
  - (it is their characterization which is not formal)
- Reasons to do this: difficult to:
  - characterize formally thousands of already existing *packages*
  - characterize formally some components (i.e., editor)
- What Amos tackles can also be seen as "how would you work if you had 15.000 components?". Searching becomes an issue — not yet solved in the components world.
- Once the package is described, it can be taken as a component

* The author has been partially supported by the EU Project IST 2001-34717 Amos. The author wishes to thank also the project participants, whom many of the ideas herein are due to.

Software development is currently one of the most important and strategic activities for any country; the creation of new software is now at the heart of many technological advances. This is not only an issue for businesses, but also for the users who request new functionalities and services, and for the governments themselves.

Many opinions have been put forward lately about the rôle that proprietary code is to play in this scenario, some of which bring about doubts as to whether proprietary software:

- can evolve fast enough as to keep pace with rapid changes in user requirements, and
- is appropriate to be used in tasks in which the security of sensible data cannot be compromised (recall that most proprietary software code usually remains largely unscrutable for the bulk of the C.S. and I.T. professionals).

Open Source Code (OSC) [Ini97], in many variants, offers an alternative to more traditional development schemes. In short (and not exhaustively), and among other characteristics, OSC development:

- is usually decentralized, in the sense than umbrella organizations do not always play a fundamental rôle;
- gives access to the source of the final products, which allows other developers to audit and reuse that code;[1]
- is often distributed at no charge for the final user (although a fee for maintenance, manuals, packaging, etc. is accepted within the community).

The rise of OSC is now materialized in several thousands of software products, ranging from smart shell scripts to whole database systems, digital image processing packages and programs, and operating systems. The quality [Whe02] and availability of many of these products has made several companies and software vendors to gear towards using and producing environments based on OSC.

One of the characteristics of OSC-based development is that the source code of all products is available, and in many cases can be studied and adapted. Therefore, it is not strictly necessary to look for, e.g., libraries which provide exactly the required capabilities: an approximate behavior is often enough, since programmers can change it. Even more, programs which are known to perform similar tasks can be inspected to locate reusable parts. Studies [WES87] show that programmers are resonably good (and consistently optimistic) at the task of deciding whether to adapt existing code or start writing from scratch.

In some (very informal) sense, source code which performs a well defined task can be termed as component, despite its behavior not being wholly specified with the accuracy level CBD needs. In order to mark this lack of formalism, we

[1] It is not always the case that OSC is free, and in this case it is usually termed *Free* or *Gratis*. We will not deal with this matter here, but we will assume that there the source code is available and reusable, without caring about its origin.

will use the name of *package* for OSCs piece, with the proviso that we are not necessarily referring to the so-called packages in many distributions of popular operating systems.

The Amos project proposes a method and a tool to characterize and systematically select among a database of package descriptions, those packages to be assembled in order to realize a software project (or a part thereof). We will describe the ideas behind the project and sketch its relationship with CBD and LP.

2 Background

Finding appropriate components is a prerequisite in order to reach the point of assembling them. Presently, the only reliable way to do this is by extensive search through software libraries. Current research on software matching is focused on two aspects:

- The use of formal languages to describe packages and to match them at the interface or functional level. This approach requires software development with a formal treatment and accurate descriptions of the software through all the process, This generates very precise matches, but it is difficult to extend to higher-level, informal descriptions, usually found when software has not been formally developed.
- The use of natural language processing to search for a specific package inside of a large repository. This is more closely related to the work proposed in this project (see for example the ROSA [dRG95,GI95] software reuse environment). It is however more focused on finding components matching a specific pattern expressed in English (e.g. "tool that searches text in files"). It is also usually based on simple single-line sentences to describe the packages, and cannot request a series of capabilities. Therefore, it is not suited to large scale components, and cannot perform a "minimum cost matching" (in terms of the extra effort needed to couple the retrieved packages).

In [MMM95], recent research work on the software matching field is analyzed and classified. Following that paper, our approach can be termed as an extended lexical-descriptor frame-based approach, where the strict tabular-based approach of adding semantics to dictionary words is extended with a more flexible ontology that is in general tree-based. Also, in the same review it is shown that a benefit of lexical based approaches is that they are capable of high recall and precision, and are not constrained by limitations of current theorem provers or formal specification methods. On the other hand, composition of packages selected using non-formal descriptions may not be immediately possible, but we are assuming the availability of code, and therefore programmers can bridge the gap between what is available and what is needed.

3 Building Software Based on Open Source Code: the Amos Approach

The core of the project is the development of an ontology for Open Source code, able to describe code assets, and the implementation of an indexical search based on the descriptions of each of the instantiations of this ontology.

The ontology provides an underlying tree structure which adds semantics (and more information) to the database contents, and instantiations of the ontology provide descriptions of packages. The dictionary is an unordered set of terms which are used to describe the items in the database of package descriptions. The dictionary is updated by adding the terms necessary to describe new items in the database, until a sufficiently rich set of terms appear in the database. Synonyms (different wordings that represent the same concept) and generalizations can be associated to any element in the dictionary. Synonyms are useful for users in whose field of knowledge a particular term is applied to some concept; generalizations are used to broaden a search when the more specific term yielded no matching description.

The search engine is in charge of answering the user request (a series of terms from the dictionary meaning desired capabilities), with a set of packages whose descriptions give the requested capabilities. The set of returned packages should cover as much as possible the user's request. In the search process, the needs of some packages (i.e., packages which in turn need other packages) are taken into account to return a set of components as self-contained as possible. It is possible, however, that some query term, either initially entered by the user or generated while performing the search, remain unmatched. Different measures of optimality can be used to choose among different search possibilities.

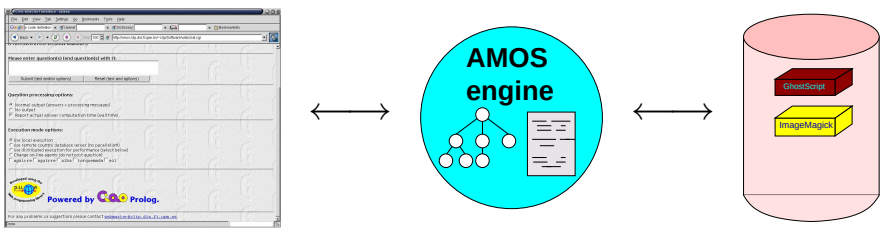

Fig. 1. A high-level view of the architecture

3.1 The Ontology and the Dictionary

The ontology is a set of structured, tree-like commented "slots" that are used to store facts about source code packages in general. This is unlike other common uses of ontologies in that we will use it not only to structure information, but also to perform reasoning (in the form of search) on them, by means of

a matching engine. Most ontology definitions are done nowadays using XML-based languages and representations like DAML+OIL [FvHH+01,HPS01]. We have chosen a different approach, because we did not foresee a lengthy work on the ontology itself and we preferred to have an easy-to-understand, easy-to-parse, human-readable description of the project. In addition to that, the reasoning part will be performed by a matching engine programmed with the Ciao Prolog [HBC+99] programming environment, with which parsing the ontology files generated withing the project would be easy.

For this reason, and to facilitate the initial work, we have adapted a very simple ontology description language that is simply an enumeration of classes. Attributes are stated in lines:

```
<x>.<y>:                <z>                        (type):a:b
```

which means that in class `<x>` the modifier `<y>` has value `<z>` and type `(type)`, and has minimum and maximum cardinalities `a` and `b`, respectively. If either `a` or `b` are not present, then absence of constraints is assumed. There is no assumption of any ordering among the ontology fields. This simplified representation allows using common Unix text tools to process it, and it is relatively easy to read and transform into other representations. Common types, such as string, date, and others, are part of already existing RFD standards (see, for example, `http://www.w3.org/TR/xmlschema-2/`, for a definition of standard datatypes).

Two fields have special importance to perform searches: the one which expresses the requirements of a package, and the one which expresses what capabilities are provided by a package. Both are simply expressed as lists of dictionary items. Initial user requirements are satisfied by selecting packages whose offered capabilities match those expressed by the user; in turn, requirements by these selected packages are treated similarly.

We have striven to be compliant with existing standards (still scarce at the moment). The most developed is the IEEE 1420.1 BIDM standard [RLIG93,RLIG95], and we tried to follow it with some extensions from the NHSE working group on Software Repositories. In particular, the fields for the certification property have been added to our package descriptions. These may be useful and applicable in specific fields like aerospace or health care.

3.2 The Search Engine

The search engine is in charge of generating assemblies of packages which fulfill a set of user requirements. A package A may need several capabilities $a, b, c, \ldots$, and provide several other capabilities $m, n, o, \ldots$; we will write this as $A^{a,b,c\ldots}_{m,n,o\ldots}$. A very schematic matching algorithm is shown in Figure 2. later! — where?

Requirements and capabilities can be associated to the *preconditions* and *postconditions* in the realm of plan generation, or to premises and consequences in the first order logic world. A noticeable difference is that in our case postconditions, or capabilities, are added to the *state of the world*, except for the case of the special `stream()` tag referred to in Section 3.1.

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

```
F := ∅                                  -- What has been fulfilled so far
P := ∅                                  -- Packages used so far
do                                      -- Invariant: R ∩ F = ∅
    select A ∈ {A_q^p | q ∩ R ≠ ∅}
    F := F ∪ q
    R := (R - q) ∪ (p - F)
    P := P ∪ A
until <no A_q^p can change R>
return (P, R)
```

Fig. 2. Schematic Search Algorithm

The `select` keyword expresses a non-determinism in the selection of packages: several choices are possible at every iteration of the loop. The intuition behind the algorithm is that as long as any element in the set of requirements `R` is satisfiable by some package in the database, such a package is (nondeterministically) selected, and the not yet satisfied requirements needed by this package are to be taken into account. Capabilities provided by selected packages are added to the set `F` of fulfilled capabilities, which performes memoization and helps to cut down search. The process may end without having matched all the requirements (either initially entered by the user, or generated by an intermediate package selection). Also, non-determinism in the `select` primitive may cause several possible outcomes: several assemblies and several unmatched capabilities.

The search is expected to use heuristics aimed at approximating some idea of optimality in the final results. For example, the user might want to minimize the number of packages |P|, the number |R| of unmatched characteristics at the end of the execution, etc. These (and other) optimality measures are often based on global considerations which need a completely generated plan in order to be applied. However, generating all plans and filtering them afterwards can be computationally prohibitive if many (e.g., thousands) of packages exist in the repository. Resorting to approximations which use *local* heuristics will improve performance, at the expense of obtaining suboptimal solutions. For example, in order to minimize the number of packages in the final result, it seems sensible to choose at each step a package $\mathtt{A}_q^p$ which reduces `R` as much as possible. This eager strategy may however yield suboptimal results, as it does not take into account the number of new required capabilities p introduced in the system.

In order to diminish the negative impact of this suboptimality, users will be presented with several assembly plans — those ranked higher. The user will then be able to select the one(s) which (s)he deems more appropriate. Additionally, the plans shown to the user will contain *explanations* about which package was selected at each stage of the search and the reasons of that selection, so that the user does not feel confronted to a *black box* whose internal underpinnings are unknown.

The basically symbolic nature of all the operations in the above algorithm, and the fact that there is an implicit non-determinism in the search, makes logic languages clear candidates to implement the above algorithm (surely in a more evolved form). The implicit non-determinism will also alleviate the work of keeping track of the non-explored branches, and resuming them when needed. We plan to implement the system using Ciao Prolog [HBC+99], which is a robust, mature, state-of-the-art, next-generation Prolog system, with a good programming environment which features an automatic documentation generator, a module system, and a rich set of libraries, including (remote) database access, WWW access and page generation, etc. Ciao Prolog is currently freely available for testing and use.

3.3 User Interface

The user interface will be based on WWW, since we want the tool to be remotely accessible. Actually, two different interfaces will coexist: one addressed to users who only want to consult the database, and another one for administrators who have to add new package descriptions to the knowledge base. In both cases the interface will be kept simple, paying special attention to its usefulness and compliance with widely acknowledged WWW standards.

The General User Interface will allow posting queries by letting the user constructing a conjunction of capabilities by selecting a set of terms from a list containing those in the the dictionary. After the search terms have been selected, (s)he has the possibility of giving hints to the search engine as to what type of selection has to be done, as mentioned in Section 3.2.

Once the search has finished, the user is presented with a series of possible package assemblies for the project to be built, including

- name of each package or asset (with links to its corresponding entry in the ontology format), and
- links to an explanation about which capabilities were needed at each stage, and which package was selected to reduce the set of capabilities.

In order to improve search results (and to give more control to the user), it will be possible to select one of the plans and restart the search from some intermediate point, adding some requirements (e.g., forcing some package(s) to be (not) included in the final assembly). This can cater, for example, for cases where the user knows about the existence of a package whose capabilities are handful for the task to be performed.

The Administrative Interface Package addition cannot be left to general users, since it can cause unneeded growth of the dictionary, and also a drift in the descriptions. Therefore, the WWW interface aimed at adding new assets to the database should be available only to some selected individuals, approved by the organization holding the database.

3.4 Issues in Building O.S. Code

- There is a great deal of O.S. code
- Reusing it is a possibility
  (cite how reusing is done by programmers, how they decide)
- Issues:
  - Constructing correct software based on pieces
    (point here: not only using code/libraries as-is, but changing code)
  - Characterization of code difficult:
    - lots of code already written
    - many *packages* difficult to formalize (e.g., editor widget?)
  - Searching in lots of code

can go to sections of code!

3.5 Characterizing Open Source Code

programmer extracts and characterizes code: what it provides and what it gives
the better structured and described, the more useful
lots of work — but many repositories also require that!
uses dictionary terms (with dictionary-approved synonyms)
packages into ontology (not really relevant for this part)
dictionary grows — reaches steady state
package addition has to be approved

4 Logic Programming and Amos

pre/post multi-headed clauses
put more conditions on it: no variables, no negation, sort of tabling, confluence, no repetition of rule applications, labeled rules.
tha algorithm — write an example
selection / search rules can be used as in SLD — only they can have heuristics (not unknown to LP: see the andorra principle)
lack of convergence?

5 Component-based Development and Amos

what to do with 10000 components
resolution with **all** the specification?
specification of a whole system (i.e.: a system to vote through SMS, saving votes in a database and validating later the results through an answer)?
this almost amounts to theorem proving
a two-stage process can be used: higher-level characterizations (à la Amos) to select packages, more directed proofs to make sure they fit together.
how to generate the high-level characterizations:

– As before
– Extract from specifications: it is a sort of abstract interpretation (in a feasible domain) of the specification formulas.

therefore, Amos, apart from a project *per se* will:

– shed light on search techniques for a class of clauses, which happen to model clearly certain pre/post conditions
– link this technique with what will probably be a need in component-based software development

6 Conclusions and F.W.

References

[dRG95] Maria del Rosario Girardi. *Classification and Retrieval of Software through their Description in Natural Language.* PhD thesis, Computer Science Department, University of Geneva, 1995.

[FvHH+01] D. Fensel, F. van Harmelen, I. Horrocks, D. McGuinness, and P. F. Patel-Schneider. OIL: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

[GI95] M. R. Girardi and B. Ibrahim. Using English to Retrieve Software. *The Journal of Systems and Software*, 30(3):249–270, September 1995.

[HBC+99] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[HPS01] I. Horrocks and P. Patel-Schneider. The generation of DAML+OIL. In *Working Notes of the 2001 Int. Description Logics Workshop (DL-2001)*, pages 30–35. CEUR (`http://ceur-ws.org/`), 2001.

[Ini97] The Open Source Initiative. The open source definition. Available at `http://www.opensource.org/docs/definition_plain.php`, June 1997. Probably under update.

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

[RLIG93] Reuse Library Interoperability Group. Model BIDM. Technical Report RPS-0001, IEEE Computer Society, 1993.

[RLIG95] Reuse Library Interoperability Group. Data Model for Reuse Library Interoperability: Basic Interoperability Data Model (BIDM). Technical Report IEEE Std 1420.1, IEEE Computer Society, 1995.

[WES87] Scott N. Woodfield, David W. Embley, and Del T. Scott. Can Programmers Reuse Software? *IEEE Software*, pages 52–59, July 1987.

[Whe02] David A. Wheeler. Why Open Source Software / Free Software (OSS/FS)? Look at the Numbers! Available at `http://www.dwheeler.com/oss_fs_why.html`, August 2002. Under constant update.