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 code which has not been formally developed, but which is interesting, is needed, or when there is a very large number 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 has a sound basis. 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

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 inscrutable for the bulk of the C.S. and I.T. professionals).

* 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.

Open Source Code (OSC) [Ini97] offers an alternative to more traditional development schemes. In short, and by no means exhaustively, OSC development:

- 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 small shell scripts to whole database systems, digital image processing programs, and operating systems. The quality [Whe02] and availability of many of these products has made several companies and software vendors to gear toward using and producing environments based on OSC.

The availability of source code in OSC makes it 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 reasonably 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 will use the name of *package* for an 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 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 repositories. 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 a formal software development and accurate descriptions through all the process. It generates very precise matches, but it is difficult to extend to higher-level, informal descriptions, usually found when software has not been formally developed.

[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.

– 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 feasible; however, assuming the availability of code, 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 idea in Amos is to take advantage of the high amount of existent OSC packages, many of them unlisted in popular software repositories,[2] and which have not been developed using formal methods. This is done through 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 repository of package descriptions. Instantiations of the ontology provide actual descriptions of packages. Terms used to describe code assets are contained in a dictionary, which is updated by adding the terms necessary to describe new packages as are added. Eventually, a sufficiently rich set of terms will appear in the dictionary which will then stabilize. The intention is that packages be described solely in terms of dictionary words, and users make queries requesting capabilities using only these terms. 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.

[2] We should highlight here those pertaining to vertical areas, exemplified by the NetLib (`http://www.netlib.org`) repository.

The search engine is in charge of answering the user requests (a set dictionary terms, 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 capabilities requested by the user. 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.

Figure 1 shows a schematic view of the whole tool: users interact with an interface to post queries and obtain results; this interface is connected to a search engine, knowledgeable of the ontology and dictionary terms, and, in turn, the search engine reads package descriptions from an external database —perhaps in one shot, perhaps retrieving information as search progresses.

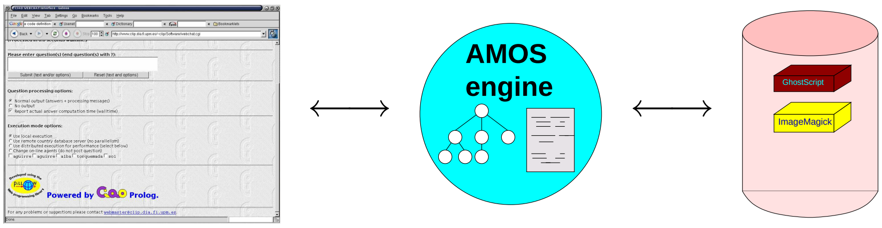

Fig. 1. A high-level view of the architecture

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.

Two slots have special importance to perform searches: the one which expresses the requirements of a package, and the one which expresses which 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 these 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 (`http://www.nhse.org/`) 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. In that code, the treatment of generalizations of terms does not appear explicitly, but, as we will see later, it can be encoded smoothly within the package description database.

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^p_q | q ∩ R ≠ ∅}
    F := F ∪ q
    R := (R - q) ∪ (p - F)
    P := P ∪ A
until <no A^p_q 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 `A` in the database, `A` is (nondeterministically) selected. The requirements needed by `A` which have not been fulfilled yet are inserted in `R`, to be taken into account in next loops. Capabilities provided by selected packages are added to the set `F`, which performs memoization and helps to cut down search. The process may end without having matched all the pending requirements (either initially entered by the user, or generated by an intermediate package selection). Also, non-determinism in the `select` primitive may generate several assemblies.

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 afterward 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 $\mathtt{R}$ 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 it (surely in a more evolved form). We plan to implement the system using Ciao Prolog[3] [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 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 construct a conjunction of capabilities selected from a set of terms in a list containing those in 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 (local) 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

[3] http://clip.dia.fi.upm.es/Software/Ciao

– 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 more requirements (e.g., forcing some package(s) to be 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 useful for the task to be performed.

The Administrative Interface Package addition should not be left to general users, since this 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.

4 Logic Programming and Amos

There is a relationship between the package description approach of Amos and Logic Programming. A package description $A^{a,b,c}_{m,n,o}$ can be read as a rule of the form

$$\underbrace{m, n, o}_{\text{Head}} \leftarrow \underbrace{a, b, c}_{\text{Body}}$$

where commas are to be understood as conjunction. I.e., the above rule is, logically,

$$a \wedge b \wedge c \rightarrow m \wedge n \wedge o$$

and search can be expressed in terms of a resolution-like method [Llo87,Rob65,Apt97]. Some points are worth noting:

– There are no variables in the literals; therefore propositional logic is enough, and carrying substitutions is not needed.
– The rule above is a multi-headed clause, and negated atoms may not appear in the body (capabilities in package descriptions always add characteristics).
– Commutativity of conjunction is assumed; therefore it is more appropriate to speak of (and use) conjunctions of literals than sequences of literals, as it is done in SLD resolution.

We will adorn the rules with the package name, as in

$$a \wedge b \wedge c \xrightarrow{A} m \wedge n \wedge o \tag{1}$$

in order not to loose the relationship between package names and their descriptions. The above rule has the implicit meaning: "assuming we have capabilities a and b and c, we can have capabilities m and n and o by using package A". This is similar, syntactically and semantically, to the interpretation of a resultant in SLD resolution [Apt97].

A search for an assembly of packages satisfying a set of initial capabilities can be reformulated as the satisfaction of an initial query with respect to a program derived from the package descriptions. However, unlike in the usual resolution procedure, which returns failure when some (intermediate) goal cannot be unified with any clause, some capabilities can remain *unmatched* in our case. This is akin to the so-called *flooding* in (constraint) logic programming. Another noticeable difference is that the (leftmost) selection rule of logic programming can not be applied, as literals in clause heads and intermediate resolvents are not assumed to have any implicit order. An answer to a query in our setting is, therefore, a set of packages plus a (possibly empty) set of unmatched capabilities. The basic resolution step can be compared with the resolution rule of logic programming, the latter being

$$\frac{\mathbf{A}, B, \mathbf{C} \qquad B \leftarrow \mathbf{B}}{\mathbf{A}, \mathbf{B}, \mathbf{C}}$$

where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are sequences of literals, with the meaning of conjunctions, and B is a single literal. In order to deal with multi-headed clauses we need literals to be explicit in the rule, as in

$$\frac{A_1 \wedge \ldots \wedge A_k \wedge \ldots \wedge A_n \qquad A_1 \wedge \ldots \wedge A_k \wedge \mathbf{B} \leftarrow \mathbf{C}}{\mathbf{C} \wedge A_{k+1} \wedge \ldots \wedge A_n}$$

Note that as several literals of a head can be chosen to be reduced in every step, the search tree has branching factor greater than in SLD resolution.[4]

In order to keep track of the package names and the unmatched characteristics, the above rule can be extended to:

$$\frac{\langle A_1 \wedge \ldots \wedge A_k \wedge \ldots \wedge A_n, \mathbf{P}\rangle \qquad A_1 \wedge \ldots \wedge A_k \wedge \mathbf{B} \stackrel{P}{\leftarrow} \mathbf{C}}{\langle \mathbf{C} \wedge A_{k+1} \wedge \ldots \wedge A_n, P \cup \mathbf{P}\rangle}$$

where $\mathbf{P}$ is the set of already used packages. The final answer is a tuple $\langle \mathbf{R}, \mathbf{P}\rangle$ with a conjunction $\mathbf{R}$ (possibly empty) of capabilities still to be fulfilled, and a set of packages $\mathbf{P}$ used within the search. A final answer is reached when no application of the reduction rule can change the accumulated set of packages —i.e., when the capabilities to be fulfilled cannot be deduced from the program. An initial query is a tuple $\langle \mathbf{Q}, \emptyset\rangle$, where $\mathbf{Q}$ is the conjunction of the capabilities the user needs.

Ideally, literals which have been proved do not have to be reexecuted again, since capabilities are monotonically added while searching. This is similar to the use of tabling [SSW93,SW92] in Logic Programming, which is exemplified in the code in Figure 2 by the variable `F`. As there are no variables, maintaining

[4] In fact, choosing single-headed (Horn) clauses for Prolog was deliberately chosen to avoid that combinatorial explosion [Col93]. We are following here a path reverse to that in history.

variant calls is not needed, and once a rule has been applied it does not need to be applied again —**if** tabling is used; Figure 4 shows an example.

Dictionary terms which have a generalization (Section 3) have a clean counterpart in the logical version of the database of descriptions. If a is a capability, and b is a generalization of it (for example, `C_standard_library` may generalize `string_processing_routines`), then the rule $b \stackrel{G}{\leftarrow} a$ models the generalization. When a is sought for, resolution will substitute it for b, the more general term. The logical view is also consistent: if b, a more general term, holds, a, more specific, holds as well, as it is implied by b. The rule is adorned so as to keep a trace of the generalization rules used.

The code in Figure 3 is a program corresponding to a possible set of packages. Rule E denotes a package which does not need capabilities (e.g., a very basic library). Figure 4 shows two different derivations, which finish with a different set of packages and a different set of unmatched capabilities. Note how in the rightmost branch, rule E was applied twice. This would not have been necessary had tabling been applied, as in that case a would not had been added again to the conjunction of capabilities.

$$
\begin{array}{ll}
c & \stackrel{A}{\leftarrow} a \wedge b \\
e \wedge f & \stackrel{B}{\leftarrow} a \wedge d \\
f & \stackrel{C}{\leftarrow} a \wedge c \\
e \wedge c & \stackrel{D}{\leftarrow} d \wedge f \\
a & \stackrel{E}{\leftarrow} \Box
\end{array}
$$

Fig. 3. A Program

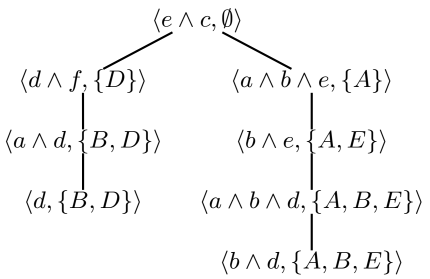

Fig. 4. Two derivations (with different answers)

Requirements and capabilities can also be associated to the *preconditions* and *postconditions* used in plan generation. This opens the possibility of using the plethora of search strategies already developed and studied.

5 Component-based Development and Amos

Besides proving correctness of composition, a problem that Component-based Development has to face is to find the right components for a task and testing that they are compatible. While this can be made by testing components one against each other, this checking is presumably very costly in computational terms, especially if the number of packages is large: each component-to-component testing can amount to proving a theorem. Besides, for components which are almost (but not completely) combinable, the matching would fail. This is right when the components are available only in binary form, but if source code is available (and modifiable), a less precise matching is actually desirable.

On the other hand, Amos focuses on a search, probably using some non-complete heuristics, where users receive information from the tool, and can provide some feedback to guide the search. The information Amos uses is assumed to describe the packages at a level less detailed than in a formal CDB setting, therefore allowing matches for packages with only an approximate behavior.

There is a synergy between both approaches, in that software pieces which have been formally specified usually include a notion of preconditions (i.e., what environment every operation assumes) and postconditions (what results the operation returns, assuming the preconditions hold). This is not unlike the formulation in (1), where some package preconditions (needed capabilities) are assumed and some postconditions (offered capabilities) are generated afterward.

In some sense, the characterizations of Amos can be seen as an approximation of a formalization, which could be obtained:

1. by hand, perusing the formal specification of package behavior, much as a programmer would reflect the code capabilities in the Amos descriptions, or, more interestingly,
2. by taking the Amos descriptions as an abstract interpretation of the formulae used to describe preconditions and postconditions of components.

Taking the second possibility would allow generating automatically (correct) descriptions using the domain in which the abstract interpretation procedure is formulated. This domain is probably far away from the one a programmer would use (both in syntax and meaning), but this is immaterial to the Amos system, as long as the semantics of the descriptions agree with the software semantics and among with themselves.

6 Conclusions and Future Work

We have reported on Amos, a EU funded project which aims at making it easier to build software based on the composition of Open Source Code. One of the difficulties in this case is finding the right pieces of code, a task which usually requires deep knowledge of many software products. The idea behind Amos is to use high level descriptions of code assets, and perform a search using these descriptions to find the set of packages which best (according to some measure) fulfills a set of user requirements.

The idea is to be materialized in a tool, composed of a user interface, a database of descriptions, and a matching engine which interacts both with the user (through the interface) and with the database, where descriptions are stored. Some degree of user interactivity is expected.

We hope the tool to be useful in organizations which keep their own database of descriptions, including not only publicly available packages, but also packages internally developed or modified. We expect also that software developed in vertical application markets, which quite often remains unknown, can be given more publicity and be more widely used.

The package description used in Amos can be assigned a logic programming counterpart, where each database description corresponds to a program, and initial user requirements correspond to queries. Accordingly, formally developed pieces of software (components) can be fit into the Amos scenario by seeing Amos descriptions as an abstract view of component descriptions.

As immediate future work, we want to make the current prototype a working tool, and investigate on search techniques and heuristics well adapted to the goals at hand. A more long-term project goal is to give Amos engines the possibility of distributing queries among a set of Amos tools, in order to take advantage of the different databases these matching engines have access to.

More information on the Amos project is available at its home page, reachable at `http://www.amosproject.org`.

References

[Apt97] K. Apt, editor. *From Logic Programming to Prolog*. Prentice-Hall, Hemel Hempstead, Hertfordshire, England, 1997.

[Col93] A. Colmerauer. The Birth of Prolog. In *Second History of Programming Languages Conference*, ACM SIGPLAN Notices, pages 37–52, March 1993.

[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.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[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.

[Rob65] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(23):23–41, January 1965.

[SSW93] K. Sagonas, T. Swift, and D.S. Warren. The XSB Programming System. In *ILPS Workshop on Programming with Logic Databases*, number TR #1183, pages 164–164. U. of Wisconsin, October 1993.

[SW92] Terrance Swift and David S. Warren. Compiling OLDT Evaluation: Background and Overview. Technical Report 92/04, SUNY Stony Brook, 1992.

[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.