Technical Report CLIP 3/02.0

The Amos Project: An Approach to Reusing Open Source Software

Manuel Carro   Carlo Daffara   Germán Puebla

August 13, 2002

Abstract

Amos is an IST-funded project that aims at facilitating the search and selection process of source code assets (either packages, programs, libraries, or pieces thereof), in order to ease its reuse. This is accomplished by designing a detailed ontology for the generic description of code artifacts, a dictionary for unambiguous expression of search terms, and a matching and searching engine implemented using Ciao Prolog. We present the motivation behind the project, the current design decisions, and the work currently underway.

Keywords: Open Source Code, Software Construction, Reuse, Search and Matching, Logic Programming, Prolog

1 Introduction and Motivation

Software development is currently one of the most important and strategic activities for any country; the creation of new software packages and tools is now at the heart of many technological advances. This is not only an issue for businesses, but also for the citizens and the users who request new functionalities and services to be delivered using software systems. However, a proper reutilization of open source code—not at the level of whole programs, but at the level an implementor perceives code—has, in our opinion, not yet been achieved.

1.1 Reusing Open Source Code Software

We believe that, by tapping in the vast world of open source, it is possible to create new software systems and improve existing ones in a way that is faster, competitive with proprietary software systems, and also safer, thanks to the availability of the full source code that allows to independently audit it, verify the absence of backdoors and security problems, and in general adapt and improve it. Also, some of the Open Source code packages are provably the best in their field, such as the NetLib (`http://www.netlib.org`) repository for numerical algorithms, and reusing them allows any company to access at a low cost a substantial software base that can form the basis for new products.

Note: In what follows we will not use the term “package” to refer exclusively to the well-known RPMs, Mandrake, or Debian packages: we will rather be referring to conceptual “units” of source code, which perform a well-defined function. For example, a suite of mathematical software may have inside packages which address matrix operations, Fourier transforms, etc. without any of them being individually shipped or separated from the whole of the software distribution.

Despite the amount of software available, some problems prevent a more widespread use. Among them we want to point out the lack of knowledge (and the lack of a homogeneous representation thereof) about which capabilities the available open source packages can provide to the programmer — e.g., which graphical routines are coded and available inside programs which perform treatment of astronomical images. Our intention is taking a step forward toward:

- describing capabilities of open source code assets, and
- helping the programmer to search through the described assets.

Our approach involves building an ontology of open source code assets and a tool which helps the programmer to select, among all the described packages, those which are more promising for developing the desired software. Should a standard for description of open software code evolve, it will also allow existing software indexes to contribute with their entries, therefore making it easier the interchange and diffusion of knowledge about open source code.

Our target is quite specific: the project focuses on **software developers**, system integrators, and, in general, experts in the field, whose dilemma is writing new code from scratch, or adapting already existing code [MMM95]. We are not addressing the (knowledgeable) end-user, who in general is already satisfied with simpler package search tools and web sites like FreshMeat, RPMfind, etc.

An especially important point will be the general concept of "assembly of packages", a set of packages that reuses as much as possible the existing software solutions, reducing to a minimum the amount of code and interface matching to do. This goal used to need a lot of knowledge, often spread by "word of mouth", about the exact capabilities of the involved packages.

1.2 Benefits from the Project

A clear benefit of the project is bringing a higher level categorization of the open source packages mentioned above, allowing for easier searches within the various options, and bringing in some of the advantages of the many research works available on ontologies [SWCP] and the "semantic web" [BLHL01]. This is, to our knowledge, the first attempt at making a general categorization of open source code which extends beyond a keyword search in a tree-shaped catalog.

The use of an automated tool like the one we propose will bring several advantages to an organization using it. Composition of open software pieces requires a deep knowledge of the capabilities of a vast amount of packages and their internal characteristics. The Amos tool will make this task easier, as the software builder can use not only her/his knowledge, but also that stored in the database, and guide the search and plan making by using different specifications and ranking criteria. Providing a wide selection base will make it possible to generate better assemblies as the number of eligible packages grows, and also taking into account the latest additions to the database.

The software builder will also be able to maintain a private database, probably enriched in order to reflect locally developed packages or, perhaps more importantly, packages which were locally updated for some particular need. Local experience can also be stored in the form of feedback and recategorization of packages, so that learned lessons will remain available for the whole organization: ontology and database scheme will store this knowledge, allowing an automated tool to use it effectively.

Other competitive advantages which an automated tool will bring about are:

- It will probably be faster than using an expert for the cases where the needed assembly is not a "standard" one, which an expert can probably retrieve immediately from memory—but in these standard cases, an expert is probably not necessary.
- It can be more reliable than an expert, especially because an automated engine can be made to generate a trace of its internals whereabouts, therefore producing explanations of the assemblies generated. These are helpful for the software builder, in order to trace the goodness of the program and to make sure the plan makes sense.
- Similar requests performed over a time span can show different results due to the evolution of the software database. This matches the dynamic style pursued by open software development.

- It can unify expertise from different knowledge areas if the database contains software from different fields. Thus, it can potentially obtain better results than those achievable by individual experts, whose knowledge is usually more focused on some area.

To finish with, one of the main advantages of our approach will come from two innovative aspects:

- Seeking the best **set of packages** whose *combination* creates the desired functionality, and
- Using more detailed descriptions for the internals of the packages, not restricted to just the externally exported interfaces, without the need of a development based on a strictly formal methodology right from the beginning.

In the rest of the paper we will describe work related to our approach, give a terse overview of the project, and explain more in depth the different project parts and their interaction. We will finish with an account of the current state of the project and future lines of work.

2 Background and Previous Work

There is an extensive literature on components, software frameworks, and such, and even some research work on the integration of legacy code into new systems. However, finding appropriate components is a prerequisite in order to reach this point. Presently, the only way to do this is by extensive search through software libraries. Current research on software matching which tries to address and solve some of the aforementioned problems 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 formalized software development methodology, and in general focuses on very low level descriptions, such as "component that performs breadth-first traversal of a tree". This generates very precise matches, but it does not extend to higher-level descriptions, like "editor component" since it is not possible to extend the traditional matching technologies, like traits, facets, or signatures to vaguely defined terms (or, alternatively, that an "editor component" has not been given a clear and broadly accepted formal definition).
- The use of natural language processing to search for a specific package inside of a large library. 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], an extensive summary of recent research works on the software matching field are analyzed and classified. Following that article, our approach can be classified 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 there is a comparison of various retrieval strategies and it is shown that a benefit of lexical based approaches like the one proposed is that they are capable of high recall and precision, and are not constrained by limitations of current theorem provers or formal specification methods. We believe that this approach is also substantially better suited to implementation using tried-and-true technologies like logic programming.

3 Project Overview

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. At the end of the project, this search engine and the ontology will be made available, together with a sample, middle-size database of packages and a WWW interface for making queries and for adding new package descriptions. The database can be used both as an initial example, for companies willing to use Amos in their work, and as a demonstration of the capabilities of the approach of the project.

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. Therefore, dictionary entries will grow quite quickly at the beginning of the project (or, in general, at the beginning of any instantiation thereof), and change very slowly in a more advanced stage. 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 matching the user input, which consists of a series of terms from the dictionary, with a set of packages whose descriptions fulfill the user query. 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. Measures of optimality can be used to choose among different search possibilities.

The user interacts with the matching engine and the database by means of an easy-to-use WWW-based interface. The interface provides access to dictionary terms, allows constructing a query, and returns the names of matching packages. Figure 1 shows a high-level view of the architecture and of the interaction among the basic parts of the system.

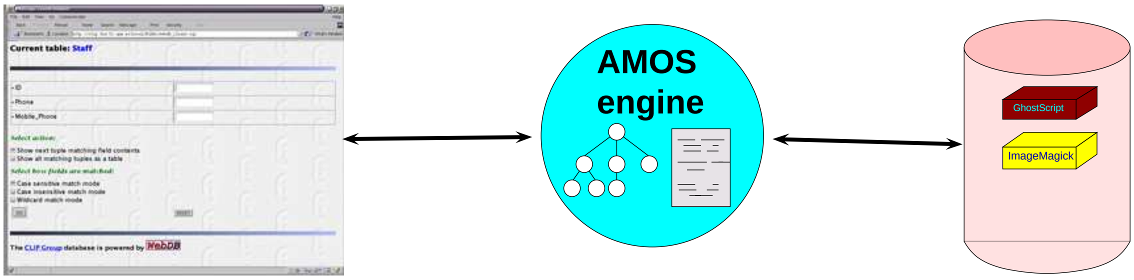

Figure 1: A high-level view of the architecture

4 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 and 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 created with the Ciao Prolog [HBC$^+$99] programming environment, with which parsing the ontology files generated withing the project would be easy.

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.

We expect the ontology to change very slowly with time. We believe that, after the first year of the project, the ontology itself will be quite complete and static. Of course, the package database and the dictionary can grow continuously, but it is unlikely that new, significant concepts will emerge after the initial work so that a change in the tree itself is required.

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. If `<x>.<y>` are repeated, it means that equilevelclasses or representation are added. 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). There are some data whose contents can be further structured (e.g., URLs), but which we will represent using a flat structure (e.g., a string), as we will process them as a whole.

As an example to illustrate the ontology description, here is a small brief from the in-progress snapshot of the ontology:

```
ontology.name:          AMOS
ontology.version:       0.2
ontology.status:        unpublished
ontology.author:        Carlo Daffara, AMOS project
ontology.authoremail:   cdaffara@conecta.it
ontology.affiliation:   Conecta srl
ontology.affiliation:   AMOS project
ontology.description:   this is the base ontology for the AMOS project.
                        it aims to capture schemas and constraints for the
                        representation of open source code packages and
                        the ancillary data about them.
ontology.comment:       work in progress
ontology.language:      english, EN
ontology.class:         asset
ontology.class:         assetElement
ontology.class:         user
ontology.class:         organization
ontology.class:         certification
ontology.class:         dictionaryItem
ontology.class:         dictionary

organization.description:  this is the main class for expressing
                           users, organizations, etc.
```

```
organization.subClassOf:   none
organization.property:     name      (string):1
organization.property:     email     (string):1
organization.property:     telephone (string)
organization.property:     webpage   (string)
organization.property:     notes     (string)

user.description:          based on organization
user.subClassOf:           organization
user.property:             affiliation (organization)

dictionaryItem.description:  holds the individual terms of the
                             dictionary, including sinonyms and
                             specializations.
dictionaryItem.subClassOf:   none
dictionaryItem.property:     dictionaryentry (string):1
dictionaryItem.property:     sinonyms        (string)
dictionaryItem.property:     generalization  (dictionaryItem)

// generalization are linked dictionary entry for relaxed attribution
// of an item. Eg. SQL database --> database

dictionary.description:      holds the entire dictionary
dictionary.subClassOf:       none
dictionary.property:         entries     (dictionaryItem):1
```

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.

Apart from fulfilling package requirements (i.e., some package might need some other capabilities in order to function properly), special attention is payed to *input/output chainability* of packages: in order to express the ability of a package output to be combined with others which require these as input, we need to be able to express package input data at the ontology level (i.e., by providing special fields, which may be empty to denote input/output data) or at the dictionary level (i.e., by selecting specific keywords to denote input/output).

We have chosen to denote input/output chaining by adding special tags to the requirements and capabilities of a package: `stream(PostScript)` in the requirements part will denote that some asset will accept a stream—in the abstract sense—of PostScript to perform some processing. This needs a special treatment during the search phase, since, in principle, these streams exist only as an intermediate data connection between two packages, while other properties are simply additive (e.g., if some package provides mathematical routines, then this capability remains present for the whole project). But, in return for that special considerations, descriptions of packages will become more homogeneous.

An example of chaining: let us suppose that we want to build a tool which converts from DVI to PDF formats. One obvious choice is to use directly the code of `dvi2pdf` which performs the translation. Another possibility is to chain the sections of code of `dvi2ps` and `ps2pdf`. Both solutions would be returned by the tool.

5 The Search Engine and Its Algorithm

The search engine is in charge of locating an assembly of packages able to fulfill a set of user requirements, expressed using terms from the dictionary. Each open source package has two main sets of terms associated: a list of capabilities which the package gives and a list of requirements which the package needs.

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. The knowledgeable reader may appreciate the similarity with *multiheaded Horn clauses*. 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 4. Therefore, a package A can need several capabilities $a, b, c, \ldots$ (i.e., preconditions), and provide several other capabilities $m, n, o, \ldots$ (i.e., postconditions). We will write this as $A^{a,b,c}_{m,n,o}$, putting preconditions at the top and postconditions at the bottom.

A very schematic matching algorithm is as follows:

```
R := <initial requirements>      -- What the user wants
F := ∅                  -- What has been fulfilled so far
P := ∅                  -- Packages used so far
do  -- Invariant: R ∩ F = ∅
   select A^p_q such that q ∩ R ≠ ∅
   F  := F ∪ q
   R  := (R - q) ∪ (p - F)
   P  := P ∪ {A}
until <no A^p_q can change R>
return P and R
```

The intuition behind the algorithm is that as long as R has any requirement satisfiable by packages in the database, a new package is (nondeterministically) selected, and the requirements needed by this package (except for those which were already satisfied) are to be taken into account. For the sake of simplicity, the algorithm above does not take into account the special `stream()` tag for data streams, which only survive one chaining, and it does not take into account generalization of packages. All capabilities the selected packages provide are incrementally added to the set `F` of fulfilled capabilities, which helps to cut down search, as only the really unfulfilled preconditions are added to the set `R` of requirements at any step. The `select` keyword expresses a non-determinism in the selection of packages, which means that several choices of packages are possible at every iteration of the loop. Also, the process may end without having matched all the requirements (either initially entered by the user, or generated by an intermediate package selection).

The selection is expected to use heuristics aimed at approximating some kind of optimality in the final results. For example, the user might want to try different heuristics for a given project, namely: minimizing the number of packages |P|; minimizing the number of programming languages in the final set of packages; minimizing the number of final features |F| which had to be fulfilled; minimizing the number of unmatched characteristics |R| at the end of the algorithm; etc.

Optimality measures are often based on global considerations which need a completely generated plan. Optimal chainings can or course be returned by selecting all possible package sets and ordering them afterward. However, it is very possible that this number is very large, and therefore computationally prohibitive, and therefore we will resort to an approximation using local heuristics. For example, in order to minimize the number of packages in the final result, it seems sensible to choose at each step a package $\mathtt{A}^p_q$ which reduces `R` as much as possible. This eager strategy might however not yield optimal results, as it does not take into account how many preconditions p are introduced in the system.

Since it is likely that several packages $\mathtt{A}^p_q$ meet the conditions imposed by a local heuristic rule, we want to make proviso for several plans to be returned, so that the user is presented with different package assemblies. All of these plans try to optimize some global optimality measure, and the user will be able to select the one which (s)he deems more appropriate. Additionally, the plans shown to the user will (optionally) contain explanations as to what package was selected at each stage, and why, so that there user does not feel confronted to a *black box* whose internal underpinnings are unknown.

The basically symbolic nature of all the operations performed, and the fact that there is an implicit non-determinism in the search, makes logic languages clear candidates to implement a more evolved form of the above algorithm. We expect that the ease of programming in logic languages (exemplified by Prolog) will be of much help in prototyping, modifying, and tuning the algorithm above until the final tool is deployed. The implicit non-determinism of Prolog will alleviate the work of keeping track of the non-explored branches, and resuming them when needed.

The Prolog system to be used will be Ciao Prolog, which is a robust, mature, state-of-the-art, next-generation Prolog system, with a good programming environment featuring an automatic documentation generator, a module system which the compiler takes advantage of to keep track of inter-modular dependencies, 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.

6 The 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 in order to build a new project, and another one for administrators who want to add new packages to the knowledge base. In the former case, special permissions are needed: making this interface completely open will create the risk of populating the dictionary with unneeded (or even inaccurate) terms, which would eventually result in lack of accuracy in the search process.

6.1 The General User Interface

The design of the user interface will be kept simple: from a high level view, the user constructs a query, hands it on to the matching engine, and examines the resulting matches.

Query construction is made by letting the user select a set of terms from a list showing those contained in the dictionary, and this set is treated as a conjunction of capabilities to be satisfied as much as possible. Each of the terms in the list will have hyperlinks explaining their meaning in order to help the user in case of doubt. This list (including the explanations) will automatically be generated from the contents of the dictionary. Additionally, choosing from a list ensures that the user does not enter "spurious" terms which are not known by the system. The alternative of natural language word-based systems (e.g., web searchers) are good to retrieve single elements (i.e., pages) when there their descriptions (i.e., the page text itself) are lengthy, but their are not appropriate for the case of shorter, terse descriptions, and they fall short when one expects the results to be somehow chained (to generate sets of packages), as requirements and capabilities are not clearly distinguished. After the search terms have been selected by the user, (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 5, several heuristics will be available.

Starting a search is done by just pressing a button. Once the search has been performed, the user is presented with a series of possible package sets for the project to be built, each of which includes, at least:

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

In many cases the search might be suboptimal, or the user might not agree completely with its results. While redoing the search with different initial terms, or with different search strategies, is always a possibility, we want to provide a means to improve the results by a semi-interactive, more elaborated procedure. The user will be able to select one of the plans and ask the matching engine to redo it until some point, select by hand one package to be included in the current plan (probably one of the candidate packages initially highlighted by the search engine), and let the engine continue from that point on. 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.

6.2 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. Apart from user certification issues, the administrative interface will offer:

- a view of the instantiated ontology package, either for editing / correcting wrong data or for entering new data,
- a view of the dictionary, in order to enter / remove terms and to edit synonym / generalization relationships,
- some integrity checks of newly entered data with respect to already existing package descriptions,
- ancillary analysis tools (e.g., determining which packages provide a certain set of description terms).

7 Conclusions, Current State, and Future Work

We have described summarily the motivation and internal structure of Amos, a project funded by the E.U. Esprit Programme which addresses the reuse of Open Source code assets. It is based on designing an ontology to describe source code; the available descriptions will be used by a matching engine, written in Ciao Prolog, to satisfy user requirements. Interaction with the system will happen through a WWW interface, also written in Ciao Prolog. We believe that the tool will be useful not only as a proof of concept, but also as a practical part of organizations whose everyday work focuses on building software projects based on creating or modifying source code pieces.

In the current state, the ontology is being finished, and a preliminary matching engine is being built in order to foresee possible scalability problems, to asses the completeness of the ontology, and to investigate feasible heuristics. We plan to develop a prototype interface in short which will allow us to introduce a preliminary set of packages and to assess the effectiveness of the technology.

Cooperating and exchanging information and experiences with other that may find our work useful, like the CoopX (`http://coopx.eu.org/`) project, is also part of the future work. We also look forward to use the ontology and results of the project as basis for a more widely available standard, and that the matching engine (which will be released under the GPL at the end of the project) will be included both in vertical, specialized sites (like mathematical software repositories, or sites for astronomical software) but also in common developer sites like SourceForge and FreshMeat.

If the same data format and engine is used at several sites with (partially) open databases, a possible future work line would be to improve the search engine in order to cooperate in a user-transparent fashion with other search engines in order to achieve more precise results thanks to the sharing of information spread across different databases.

Acknowledgments

The authors have been partially supported by the EU-funded IST Project 2001-34717 Amos.

References

[BLHL01] T. Berners-Lee, J. Hender, and O. Lassila. The Semantic Web. *Scientific American*, May 2001. Available from `http://www.sciam.com/2001/0501issue/0501berners-lee.html`.

[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.

[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.

[SWCP] The Semantic Web Community Portal. Markup Languages and Ontologies. Available from `http://www.semanticweb.org/knowmarkup.html`.