Made in MAKAO

Forum demonstration proposal

Bram Adams*

Bram.Adams@UGent.be
GH-SEL, INTEC, Ghent University
Sint-Pietersnieuwstraat 41, 9000 Belgium
Tel.: +3292643318
http://users.ugent.be/~badams/makao

ABSTRACT

Besides source code, a software system consists of a myriad of other artifacts, of which the build process has a prominent role. Without it, the system is useless. This means it needs to evolve in parallel with the source code in order to build, and hence deploy and test, the whole system. As such, the build system closely mimics the architecture of the system under consideration. Yet, as it lies at the meta-level, it has to deal with totally different, lower-level (build) concerns. And here lies the catch: little tool support exists to help people bridge the gap between the source-code dimension and the build process. We present MAKAO (Makefile Architecture Kernel for Aspect Orientation), a re(verse)-engineering framework for build systems. Its goals are to represent a build's dependency graph in a digestible way, to enable flexible querying of all build-related data and to allow efficient re-engineering of the build and even of configuration scripts. For the latter, we applied the aspect-oriented notions of join point, advice, pointcut and weaving to the build domain. MAKAO can deal with a wide range of build-related problems, but this demo focuses on the integration of new tools in a legacy system.

1. PROBLEM SPACE

Until 1977, ad hoc build and install scripts were used to automate the build process of software systems. Feldman changed everything with "make" [1], which turned out to be the most influential software build tool ever. He proposed to declaratively specify the dependencies between targets (executables, object files, libraries, source files, etc.) in so-called "makefiles", where the recipe to build a target is written as an imperative list of commands and macros. Figure 1 shows an example makefile snippet. On line 1, a makefile variable named `make_OBJECTS` is defined as a list of required object files for use in the dependencies of a target called `make$(EXEEXT)` in the rule on line 3.

Makefiles run on top of an interpreter relying on the simple observation that a target T only needs to be (re)built by its recipe if at least one of its dependencies is newer. This mechanism is transitive: for each dependency D of T, a corresponding rule is searched where D is now the target. The resulting scheme greatly improves incremental compilation of software projects and the quality of builds. If one or more of the dependencies of the rule on line 3 in Figure 1 do not exist or are newer than target `make$(EXEEXT)`, the build recipe on lines 4–7 will be executed to create an up-to-date version of this target.

Later on, portability of software required configurability of both source code and build scripts. Figure 1 illustrates this with variables

```
1 make_OBJECTS = ar.o arscan.o \
                 commands.o dir.o ... hash.o
3 make$(EXEEXT): $(make_OBJECTS)
        @rm -f make$(EXEEXT)
5       $(LINK) $(make_LDFLAGS) \
                $(make_OBJECTS) \
7               $(make_LDADD) $(LIBS)
  ...
```

Figure 1: Example makefile.

`$(EXEEXT)`, `$(LINK)`, etc. that represent platform-specific characteristics like the extension of binaries, the name of the linker program used, etc. A summary of the resulting system can be found in Figure 2. Both the (re)source(s) and build scripts are in fact templatised to abstract away from platform-specific configuration issues. The right value for these options is determined by configuration scripts written e.g. using GBS (GNU Build System)[1]. In the most general case, these generate the actual build scripts that will perform the ground work as well as fill in the remaining gaps in the (re)source(s). The combination of all this is called the build system.

Build systems play a crucial role, as various stakeholders interact with it, each with their own concerns and problems:

developers Assess the effects of their code or, if the build did not succeed, try to find out what error was the culprit. When adding new source code, they want to find out where they need to change something.

maintainers Want to learn the inner mechanics of a new system, check if there is dead code, profile things, find recent additions, etc.

deployers Try to find out what library dependencies are needed to compile and run the software.

QA division Wants to add feature and regression tests and run them as easily as possible.

researchers Try to quickly integrate experimental tools.

As a consequence, the build process implicitly contains valuable information about all facets of software, enhancing the data gained by existing reverse-engineering techniques for source code. In fact, all people interacting with the software system from the design phase on will have to deal with the build system at some time. This means that adequate tool support is required to cater for all those stakeholders' needs. More specifically, both reverse-engineering and re-engineering features are needed. User-level concerns like "what error did halt the build" or "find dead code" need to be

*Both contact person and presenter.

[1] `http://sources.redhat.com/autobook`

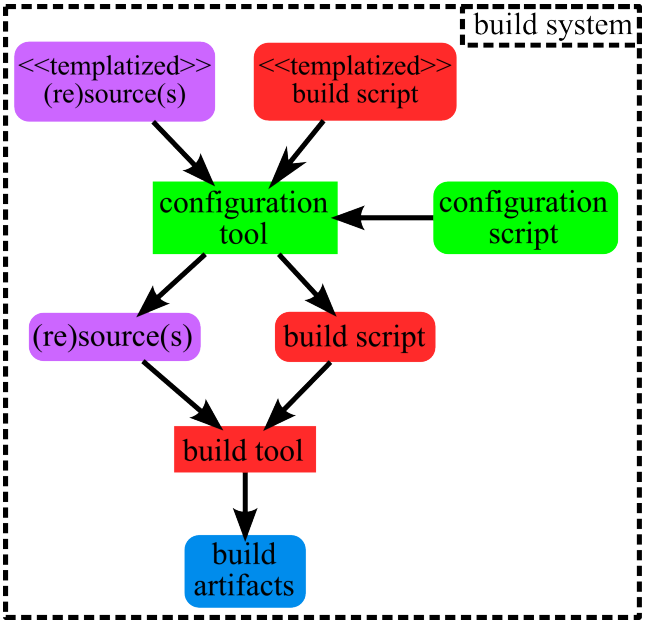

Figure 2: High-level view of build systems.

mapped on build concerns like targets and build recipes in possibly templatised build scripts. Likewise, assistance is welcome to facilitate build modifications in the event of e.g. source code refactoring.

These observations led us to the idea of an extensible visualisation and re(verse)-engineering framework for build systems, which we named the Makefile Architecture Kernel for Aspect Orientation (MAKAO).

In the remainder of this proposal, we will explain the relevance of MAKAO to AOSD (section 2), elaborate on its design (section 3) and implementation details (section 4). We will give an outline of the demo we propose (section 5), followed by a discussion of related work (section 6). Finally, we list our presentation requirements (section 7) and conclude (section 8).

2. RELEVANCE TO AOSD

There are three reasons why we named our tool suite the "Makefile Architecture Kernel for Aspect Orientation" (MAKAO) and not just MAK. First, AOSD is all about separation of concerns (SoC), be it at the requirements gathering level or e.g. at build-time. At the latter level, we are talking about typical build concerns like binaries, libraries, Java files, etc. As build systems are very complex and consist of hundreds of text files spread across a similar number of locations, the visualisation and exploration of these concerns becomes an important task, especially when one takes into account the close relationship with source code. Badly modularised build scripts severely constrain software evolution, so detecting and identifying critical spots is of utmost importance.

On the other hand, it is also interesting to explore the usefulness of an aspect language for build scripts to accommodate re-engineering, i.e. some sort of "AspectMakefile". Interesting join points are targets, dependencies and commands in build recipes, whereas useful advice can be the removal or addition of rules and/or dependencies, modifications of build recipes, etc. When e.g. some tool is to be replaced by another one in a context-dependent way and its usage is spread across dozens of rules, a simple query (pointcut) on the dependency graph can pinpoint the right targets where commands need to be modified. Then, weaving takes place both logically (modifying dependency graph) and physically (altering build scripts or even configuration scripts).

Finally, we would like to point out possible usage of MAKAO for aspect mining. We haven't looked into this yet, but we suspect that e.g. fan-in analysis [3] could be applied to the build process's dependency graph to identify possible crosscutting targets (components). Of course, it remains to be seen whether the mined information would not be too coarse-grained.

3. UNIQUENESS OF MAKAO'S DESIGN AND IMPLEMENTATION

MAKAO's philosophy is to make the implicit explicit. The declarative nature of dependency specifications is a good thing, but these are typically spread over hundreds of files and composed in some unclear way (e.g. recursively [4]). An easily visualisable representation combined with a powerful querying facility were two important requirements for MAKAO. However, more invasive problems like the addition of new tools (see sections 2 and 5) also require re-engineering of the build system. This involves e.g. introducing new build targets, adding extra dependencies to existing targets or modifying targets' recipes. The design of MAKAO needs to take both the reverse as well as the re-engineering functionality into account, unlike e.g. Ant Explorer and the BTV Toolkit (see section 6).

Knowing all this, we opted for a Directed Acyclic Graph (DAG) as the underlying data model of a build run (i.e. a concrete build), based on the following observations:

- DAGs form the underlying model of "make" [1], and hence of most of its successors. Nodes are build targets, while edges represent dependencies between two targets.
- Graphs have a natural visual representation and can be easily modified.

Providing only static views of a build system would not be useful, as the templates are too general and hence too hard to navigate or understand. As stakeholders are mostly confronted with instantiated, platform-specific build scripts and code, we chose to process dynamic traces of concrete builds, but with links back to the static build data (e.g. the commands dictionary in section 5).

On a higher level, we conceptually subdivided MAKAO into the following four components:

Explorer (Visually) Explore a representation of the build system.

Finder Query for targets, dependencies and commands based on properties.

Adviser Write modifications for targets, their dependencies and/or recipes.

Weaver Apply modifications both logically (in-memory) and physically (build and configuration scripts).

4. UNDERLYING TECHNIQUES AND TECHNOLOGIES USED

Basically, while performing a typical build, the build tool's internally constructed dependency graph is extracted. In practice, this can be obtained by using a modified "make" like "remake" or "BTV Toolkit" (see section 6) or e.g. by capturing and post-processing the debug output produced by the build tool. Currently, we use the latter option, and we implemented some converter scripts to extract the right data from the obtained build trace.

Figure 3 shows such a dependency graph for a full build of Aspicere [6], as seen in MAKAO's main panel. We implemented MAKAO on top of GUESS[2], a graph exploration tool with an em-

[2] `http://graphexploration.cond.org`

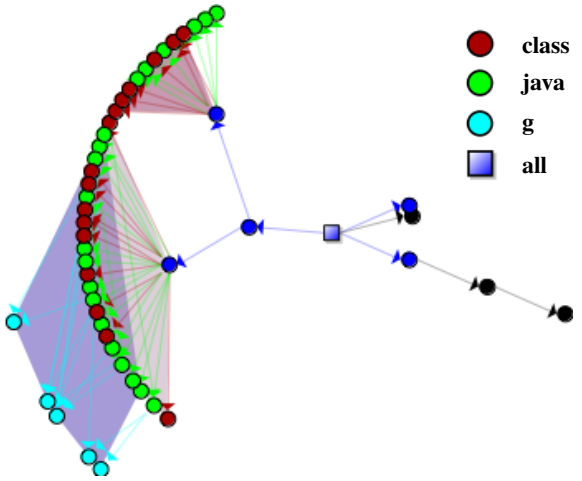

Figure 3: Dependency graph of build of Aspicere.

bedded Jython-based scripting language (called Gython). All nodes, edges and hulls are objects with their own user-definable attributes (name, concern, line number in makefile, etc.), which enables easy manipulation, navigation and transformation of the graph. One can load scripts or interactively try out some commands in the scripting console (not shown in Figure 3).

After loading, we assign a color to all targets (nodes) based on their concern. An additional "legend" panel, pasted on Figure 3 here for the reader's convenience, outlines all known build concerns in use (".java", ".o", etc.). To correlate the various targets, a colored hull is drawn around all targets of a given makefile, unless the hull degenerates to e.g. one node. Unknown concerns are colored black by default[3].

5. DESCRIPTION OF THE DEMO

During the demo, we want to show how MAKAO should typically be used by applying it on an issue we encountered during a reverse-engineering experiment [6] using Aspicere, our aspect language for C. To weave a tracing aspect into the code base, we had to integrate Aspicere's weaver into the build system to let it preprocess each source file right before its compilation. Doing this by replacing the relevant commands by a wrapper script around all compilers and source processors did not resolve subtle issues like two or more wrappers invoking each other. A simple regular expression find-and-replace script did not suffice either due to irregularities in the usage of variables, commands, comments and white-space. Manually verifying and correcting the makefiles afterwards seemed inevitable back then, but we will now revisit this case with MAKAO.

First, we perform an unaltered build while we extract the constructed dependency graph. Figure 4 shows the generated build trace, while Figure 5 reveals the .gdf graph file extracted from it. After loading this graph into MAKAO, the Explorer-component (Figure 6) allows us to verify that there are indeed C source files, but also ec-files (Embedded SQL), reports, etc. To give an idea of MAKAO's versatility, we can use a short query to highlight all paths in the build where an error occurred and we can easily identify all offending targets (Figure 7). For the graph shown, they correspond to unused components that have not been removed yet from the system.

[3]As an aside, the dark blue targets of Figure 3 really represent the same concern as the square (starting) node's one.

Getting back to the main discussion, we are interested in those targets `T` directly depending on (i.e. processing) C files and we want to insert calls to Aspicere's weaver in `T`'s recipe right before the (sole) source-processing command. To accomplish this, we will write some aspects. We write the following queries for the Finder-component (part A of Figure 8):

```
T_list=(concern=="c").inEdges.node1.findNodes()
base=[(command,tool,T) for T in T_list
                       for command in commands[T.name]
                       for tool in ["gcc","CC"]
                       if command.find(tool)!=-1 ]
```

On line 1, we select all source nodes of edges leading to C targets[4], presuming that there are no duplicates (to keep the demo more focused). Then we try to find each `T`'s sole source-processing command in its recipe using a Gython list comprehension. As mentioned in section 3, the commands dictionary links back to the commands found in the static representation of the build system. The tools we mention in this query (line 4) could have been discovered earlier on by querying too. Now, we can write advice for the Adviser (part B of Figure 8):

```
before_advice=
     ["\n".join([c.replace(t,t+" -E -o ${<}"),
                 "aspicere.sh ${<}"])
           for (c,t,ta) in base                ]
```

The `join` function concatenates invocations of the selected tool `t` (in preprocessing mode) and of Aspicere's weaver to get the desirable effect. Finally, the Weaver should weave all these concatenated commands at the right place `c` in the recipes of the targets `T` as before advice, both in MAKAO's memory representation as in the proper build and/or configuration scripts (part C in Figure 8):

```
cc_weaver=weaver("aspicere-cc",1)
cc_weaver.weave_before([T for (c,t,T) in base],
                       [c for (c,t,T) in base],
                       before_advice)
```

For demonstration purposes, we pick out two targets of one particular build script (Figure 9) that each will be affected by another aspect (one for the C files, another one for Embedded SQL files). Their build recipe before logical weaving is shown in Figure 10. Now, we perform the (logical) weaving (Figure 11) and check back the woven build recipes of our two reference targets (Figure 12). We see that the in-memory build recipes have been updated with extra lines marked with special comments. To verify things, we could now issue some new queries on MAKAO's memory model to see whether any relevant targets were skipped.

In order to do the physical weaving (Figure 13), we use two Perl scripts that were generated during logical weaving. Figure 14 shows the build script of Figure 9 after physical weaving. The makefile rules have been modified as intended. If desired, we can also unweave aspects. Figure 15 e.g. shows the build script of Figure 14 after unweaving the aspect for Embedded SQL files.

This demo illustrates how MAKAO could have helped us tremendously doing our case study back then, but also that it is flexible enough to be used in a whole range of other applications.

6. RELATION TO OTHER INDUSTRIAL OR RESEARCH EFFORTS

Personally, we got involved in research about build systems when applying Aspicere, our aspect weaver for C, on an industrial code base [6] (see section 5). We spent a great deal of our time finding

[4]In the same vein, we also wrote an analogous aspect for embedded SQL files (.ec).

out the right place to insert our tools in the build system, writing an ad hoc makefile transformer and manually verifying the transformation results. Detecting all source code processing tools also required scanning manually through the whole build trace.

Other people have done dome research about build systems before. Qiang Tu and Michael W. Godfrey [5] have proposed the build time architectural view as an addition to Kruchten's "4+1" view model [2], together with a proper notation. They backed this claim by identifying a new architectural style found e.g. within GCC[5] and the Perl interpreter[6]. The Build Time View (BTV) Toolkit[7] is a tool kit they developed to interpret the build-time architecture of a system. It is merely targeted at visualizing the build system, without support for any modification or manipulation.

yWorks' Ant Explorer[8] is similar in this regard, but targeted at Ant[9] processes. Finally, we mention Remake[10], this is an improved GNU Make with extra tracing capabilities and even a dedicated makefile debugger. One can set breakpoints, step through the build, etc.

We can remark that MAKAO is the only tool allowing manipulation of dependency graph and/or build scripts.

7. HARDWARE AND PRESENTATION REQUIREMENTS

We only need a beamer with good color support (see Figure 6), as we bring our laptop with us.

8. CONCLUSION

Build systems are inherently part of and coupled to software systems, and they offer valuable architectural information to various stakeholders. To facilitate quick understanding and clever modifications, MAKAO offers visualisation, querying and flexible manipulation of both build structure and behaviour, inspired by aspect-oriented techniques. This demo shows MAKAO in action while integrating new tools into an existing build system, identifying erroneous build paths, etc.

Acknowledgements

The author wants to thank Kris De Schutter and Andy Zaidman for their support.

9. BIBLIOGRAPHY

[1] S. I. Feldman. Make - a program for maintaining computer programs. *Software - Practice and Experience*, 1979.

[2] P. Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.

[3] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *WCRE*, pages 132–141. IEEE Computer Society, 2004.

[4] P. Miller. Recursive make considered harmful, 1997.

[5] Q. Tu and M. W. Godfrey. The build-time software architecture view. In *ICSM*, pages 398–407, 2001.

[6] A. Zaidman, B. Adams, K. De Schutter, S. Demeyer, G. Hoffman, and B. De Ruyck. Regaining lost knowledge through dynamic analysis and Aspect Orientation - an industrial experience report. In *CSMR*, 2006.

[5] `http://gcc.gnu.org`

[6] `http://www.perl.com`

[7] `http://swag.uwaterloo.ca/~xdong/btv`

[8] `http://www.yworks.com/en/products_antexplorer_about.htm`

[9] `http://ant.apache.org`

[10] `http://bashdb.sourceforge.net/remake`

Figure 4: Small part of the generated build trace.

Figure 5: Extracted dependency graph.

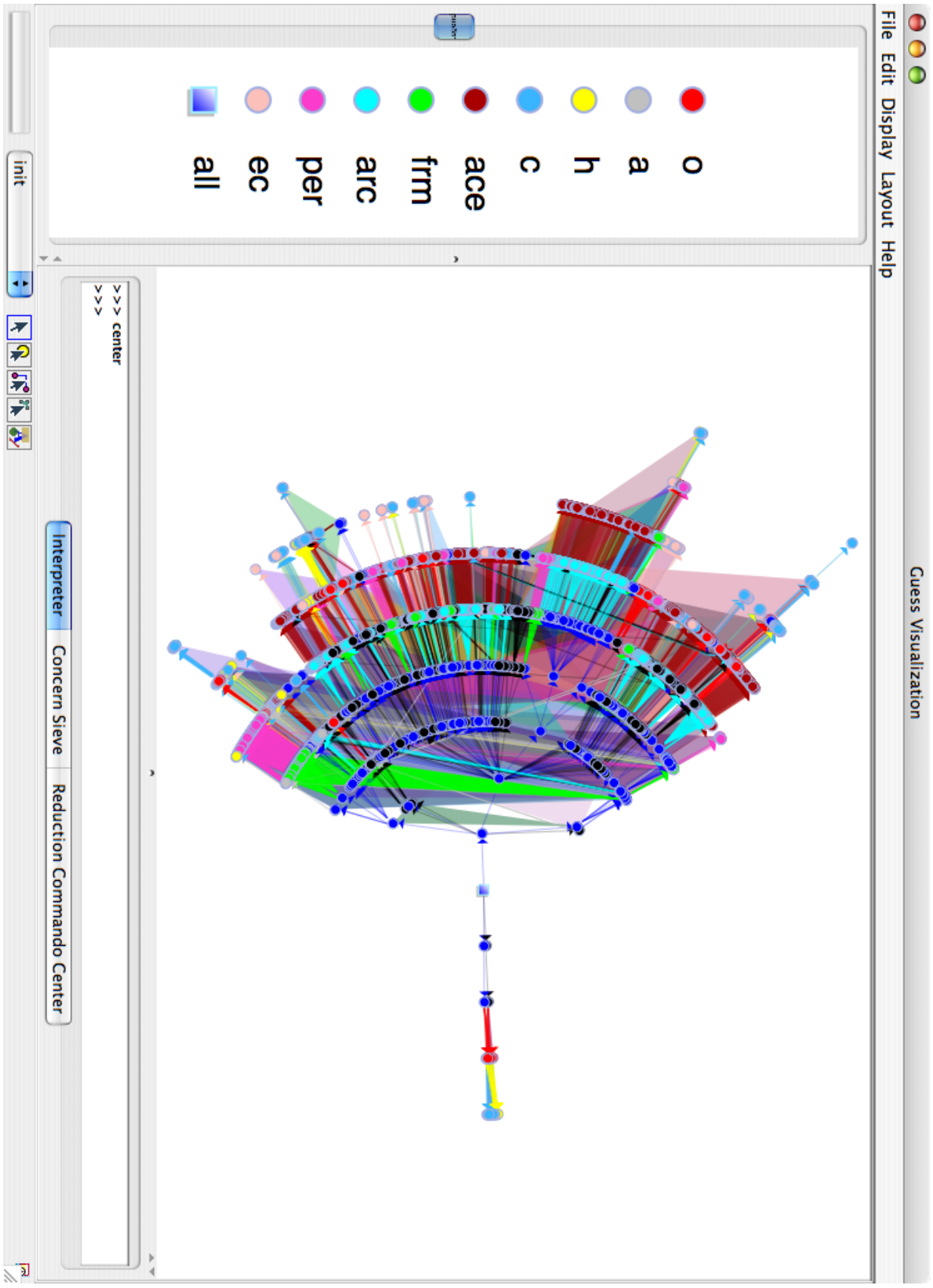

Figure 6: Full dependency graph opened in MAKAO.

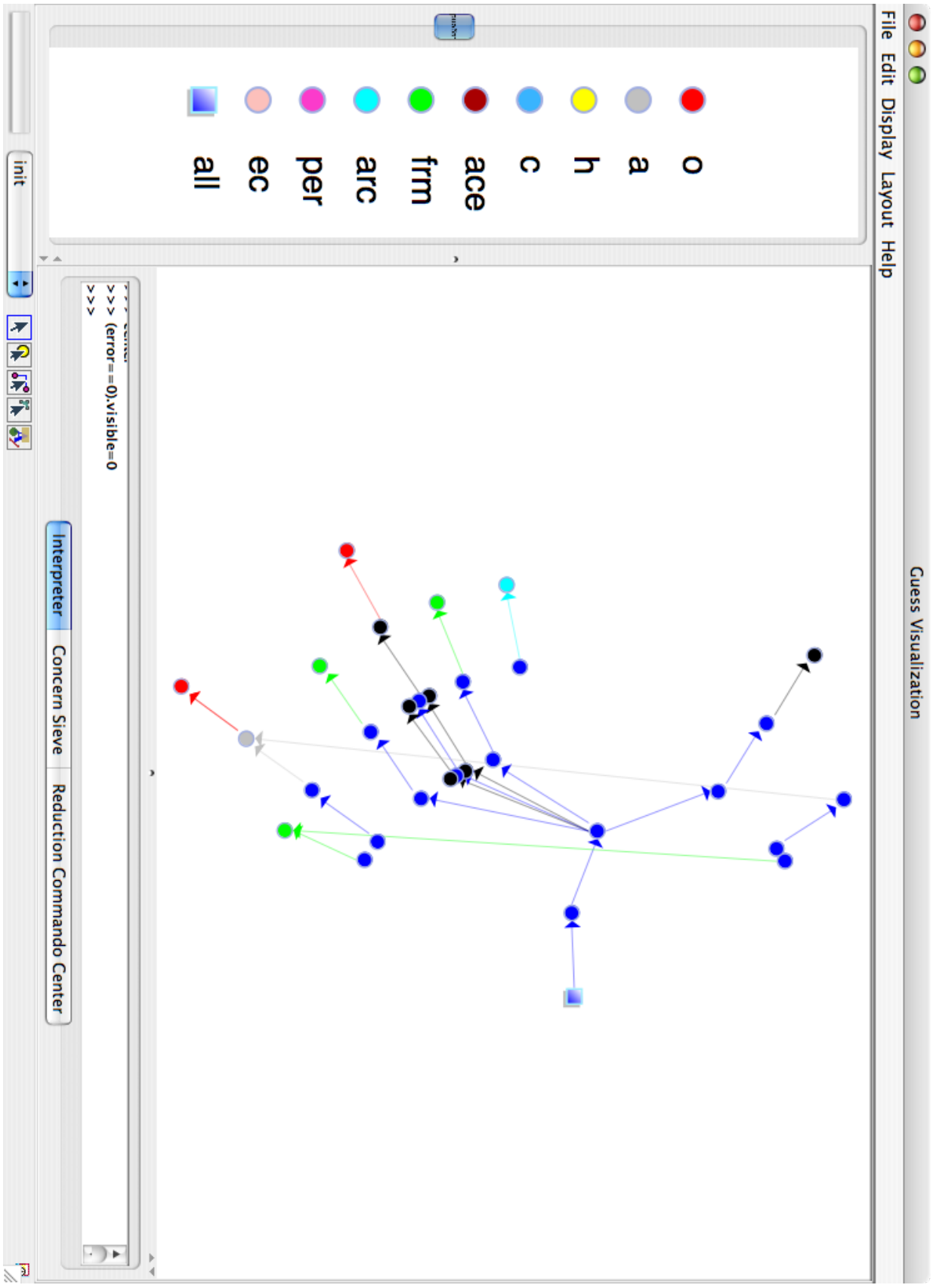

Figure 7: Subgraph of dependency graph corresponding to erroneous build paths.

Figure 8: Aspect for C files. The parts labeled "A", "B" and "C" correspond to the code snippets of section 5.

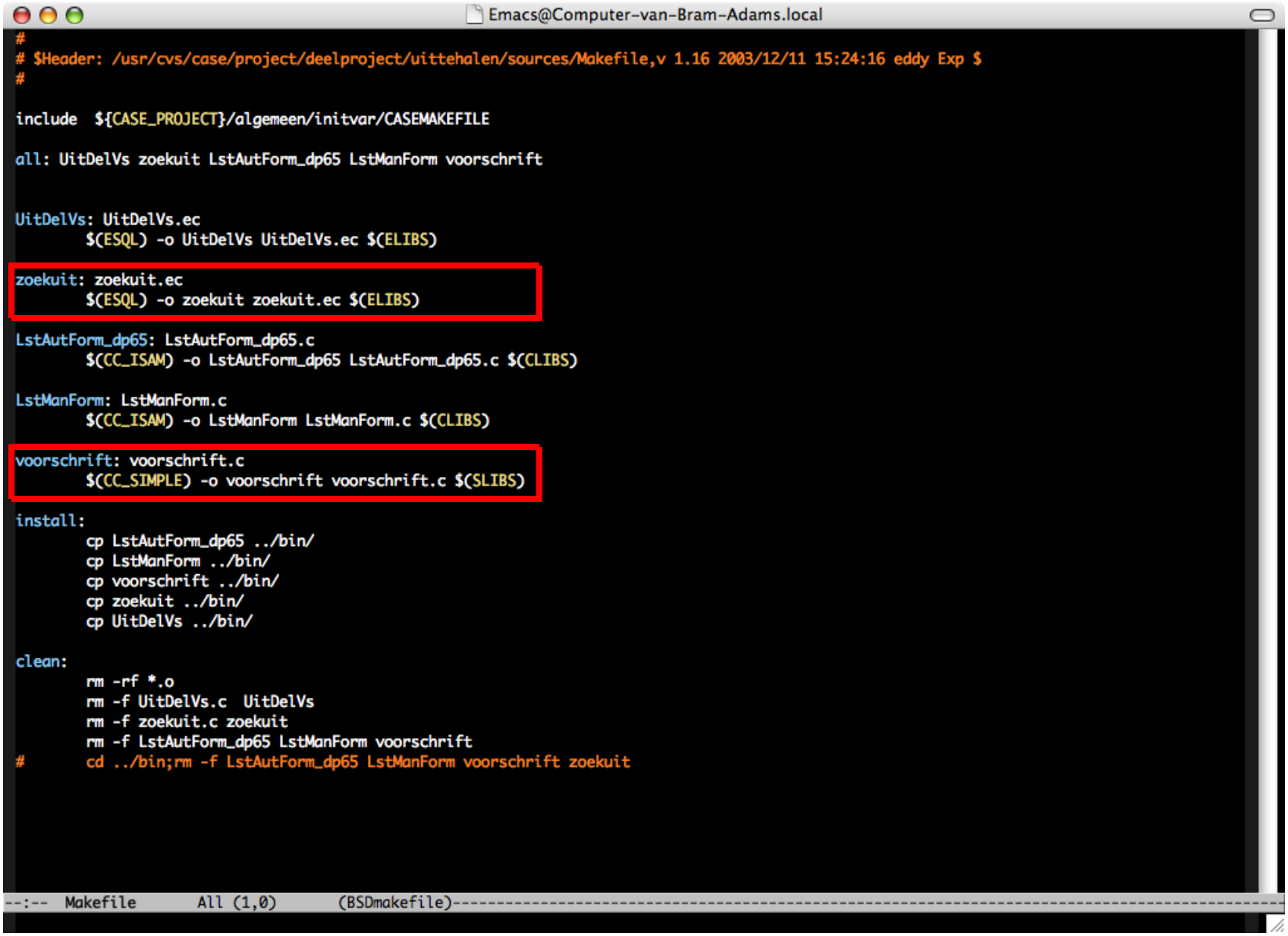

Figure 9: Makefile containing the two reference targets.

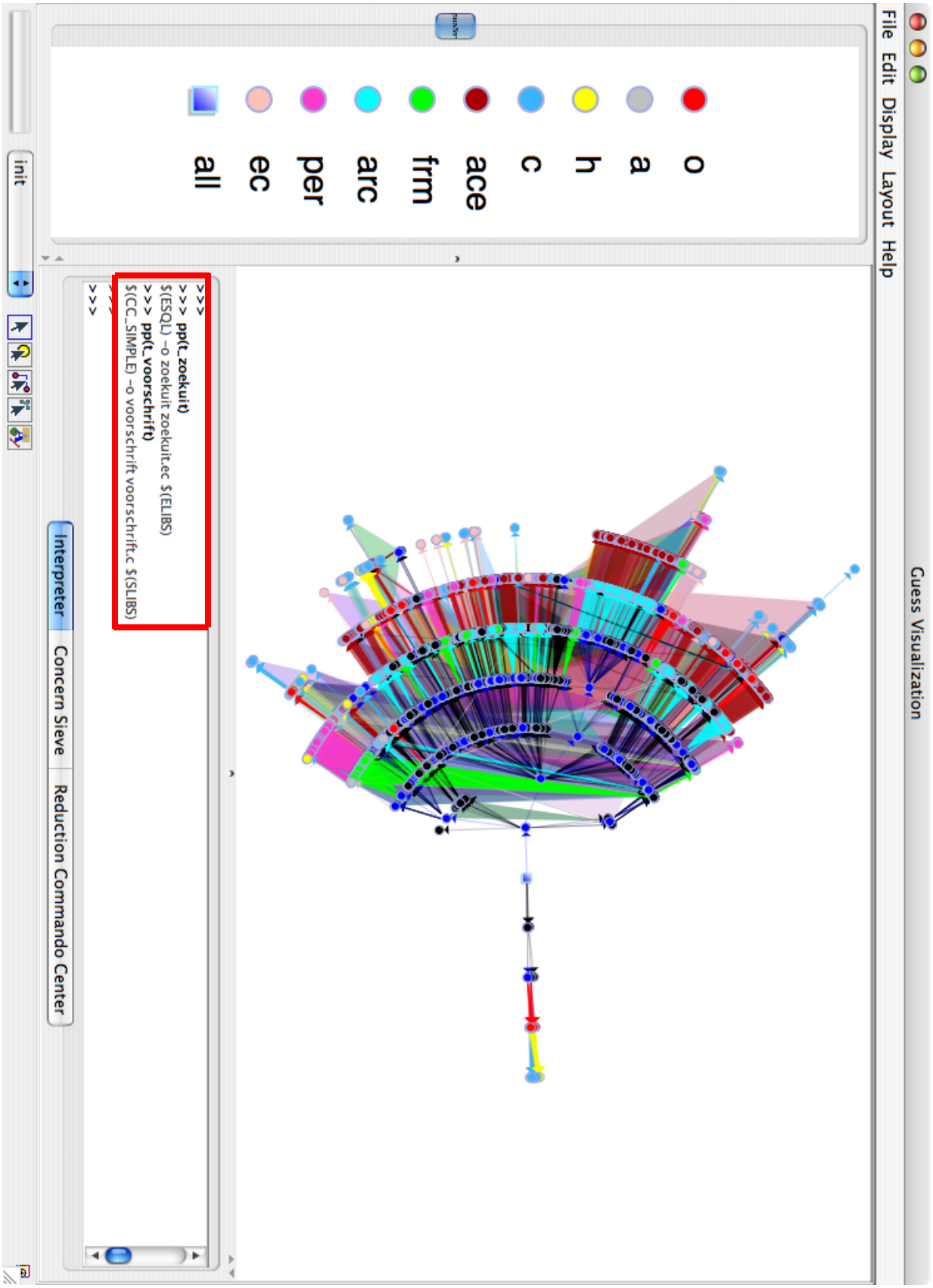

Figure 10: Original build recipes of the two reference targets as seen in-memory.

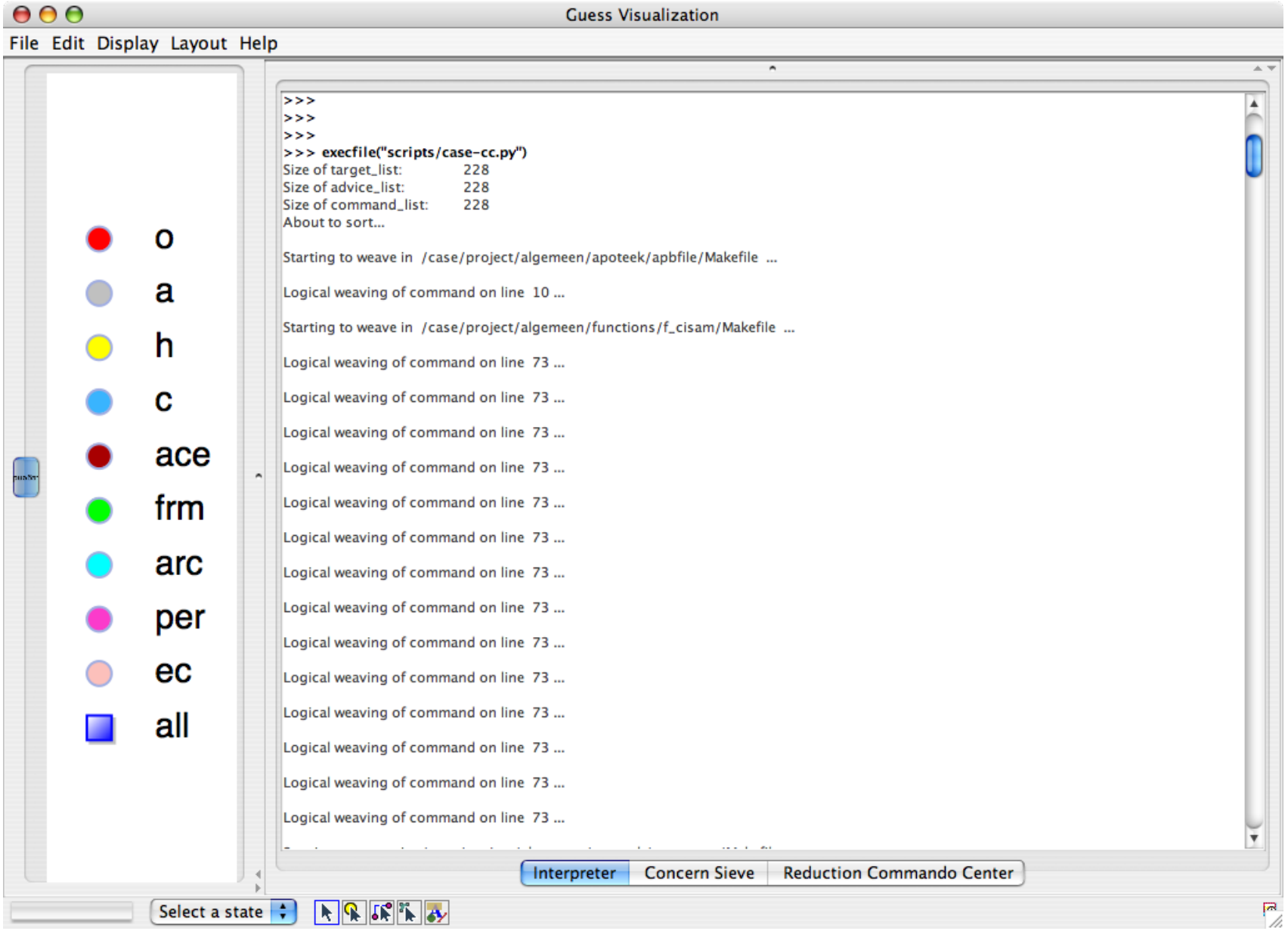

Figure 11: Invoking the logical weaving.

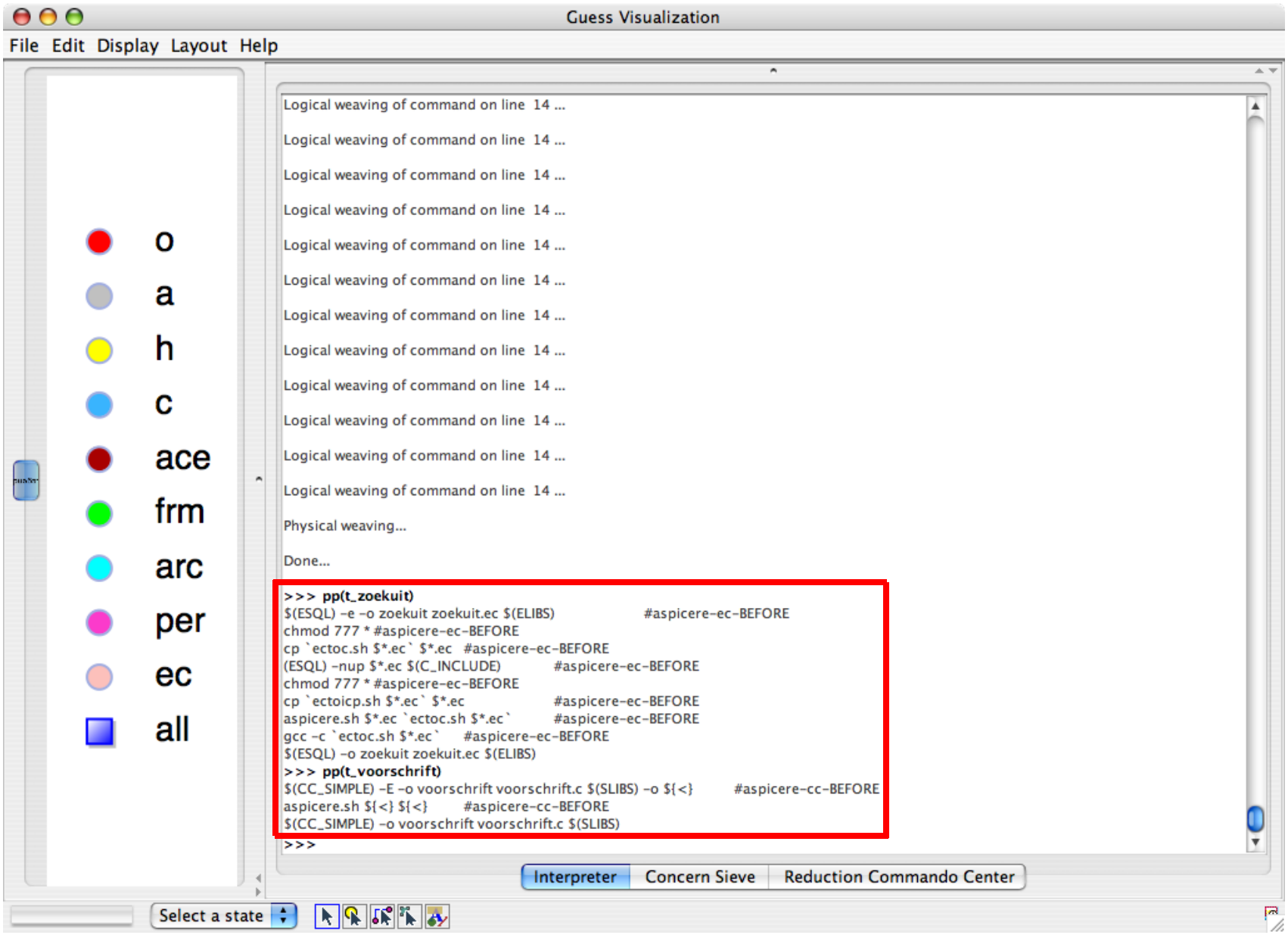

Figure 12: Modified (in-memory) build recipes of the reference targets.

Figure 13: Running the physical weaving scripts.

Figure 14: Example makefile after physical weaving of both aspects.

Figure 15: Example makefile after unweaving of aspect for Embedded SQL files.