Design recovery and maintenance of build systems

Bram Adams, Herman Tromp
GH-SEL, INTEC, Ghent University
Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium
{Bram.Adams,Herman.Tromp}@UGent.be

Kris De Schutter, Wolfgang De Meuter
PROG, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
{kdeschut,wdmeuter}@vub.ac.be

Abstract

The build system forms an indispensable part of any software project. It needs to evolve in parallel with the source code in order to build, test and install the software. Unfortunately, little tool support exists to help maintainers gain insight into the build system, much less to refactor it. In this paper, we therefore present the design and implementation of a re(verse)-engineering framework for build systems named MAKAO. At its heart the framework makes the build's dependency graph available in a tangible way. Aside from visualisation, this enables powerful querying of all build-related data, as well as various filtering techniques to define views on the build architecture. If desired, all this gathered information can be put to use to write aspects for refactoring the build. Afterwards, validation rules can help in assessing failure or success. We applied our implementation on an industrial C system and the Linux 2.6.16.18 kernel, with good results.

1. Introduction

Up until 1977, ad hoc build and install scripts were used to automate the build process of software systems. Then, Feldman introduced a dedicated build tool named "make" [9]. Its innovation was the explicit declarative specification of the dependencies between targets (executables, object files, etc.) in textual "makefiles", together with a "recipe" (an imperative list of shell commands) to build a target. "Make"'s time stamp-based updating algorithm considerably improved incremental compilation of software projects and also the quality of builds.

Soon after build tools became more or less a commodity, portability of software across various platforms became an important concern. Include directories or compiler versions vary between systems, so source code and build scripts need to be configurable. This means that they contain parameters, specified in configuration scripts, which are resolved on the system on which the build will be performed.

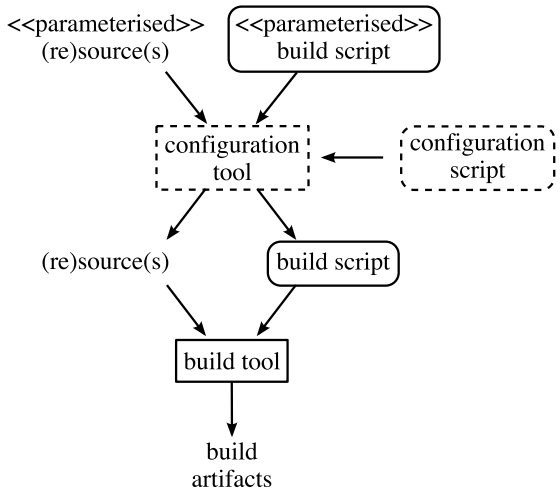


Figure 1. High-level view of build systems.

Having a Figure 1-like build system in place, each non-trivial source code modification may potentially break the build, e.g. when files are moved, added or deleted. One can only avoid this by evolving the build system in parallel with the source code, but this requires thorough knowledge of the build's internals. Recently, the KDE desktop environment project has switched from a GNU build system (GBS) to cmake for exactly this reason.

Despite lots of re(verse)-engineering approaches geared towards source code, to date little attention has been devoted to the re-engineering of a build system. This is unfortunate, as build systems can tell us a lot about a project's development architecture [15]. We therefore present the design and implementation of a visualisation and re(verse)-engineering framework for build systems named MAKAO. Its core functionalities are the visualisation, querying, filtering, refactoring and validation of build systems.

This paper contributes:

- a rationale of, and requirements for a re(verse)-engineering tool for build systems,
- the design and implementation of such a tool, and

- the application of this tool to some typical build problems in two software projects.

We will first elaborate the need for a re(verse)-engineering tool (section 2), from which a number of requirements can be derived (section 3). After discussing related work (section 4), we show how the requirements steered the design and implementation of our tool, MAKAO, in section 5. Section 6 applies MAKAO on various build problems in an industrial C system and the Linux 2.6.16.18 kernel. Section 7 discusses future work, while section 8 presents the conclusions of this paper.

2. Problem statement

Various stakeholders interact with a build system, each with their own concerns and problems. Developers e.g. assess the correctness of their code and, if the build failed, try to find out the cause (e.g. missing dependencies). When adding new sources, they need to understand how to change the build. Maintainers on the other hand require full knowledge of the inner mechanics of a system [7], want to check if there is dead code, profile, check recent changes, etc. Deployers prepare and configure the environment (library dependencies, system variables, etc.) to compile and install the software, while Quality Assurance just wants to add and run feature, regression and integration tests as seamlessly as possible. Researchers are interested more in (un)plugging experimental tools in a software system, but for this they need to grasp the development architecture.

In fact, all people interacting with the software system from the design phase on (except for end users) will have to deal with the build system at some time. This means that, given tool support, a lot of useful data can be mined from it. The recovered knowledge can serve both for comprehension [11] as well as for re-engineering purposes.

Historically, build tools have suffered from a lot of problems hampering understandability and maintainability. "Make" and its direct derivatives attached semantics to syntactic elements like tabs and spaces, and also made it easy to access non-portable shell scripts and commands. Worse, whereas the specification of dependencies is a good thing, manually managing them is tedious and error-prone. Even the use of dependency generators like mkdepend or newer build tools like SCons does not conceal the real cause: the search for build script modularity. Software projects normally define components and layers to distribute functionality and effort. This implies that having only one global build script is not feasible and that a more modular approach is needed, mimicking in fact the source code architecture. In the UNIX world, this problem is usually solved by means of the so-called "recursive make"-technique [16]: a build script in the top-level directory invokes build scripts in each subdirectory and so on until the whole system is built.

In [16], a serious conceptual mistake was exposed with recursive "make": it cannot handle dependencies across "make" (sub)processes. Consider the following situation:

- Targets D and E depend on some file F, but live in different directories and, hence, different makefiles.
- A first "make" subprocess builds D.
- A second subprocess changes F and then builds E.

Clearly, target D should be remade too, as its dependency F has been updated. The recursive build prevents "make" from detecting this, because each "make" process knows only about its own targets and dependencies. If the rules for both D and E were in the same makefile (process), "make" could choose a better evaluation order. Traditional solutions to this visibility problem include sorting subdirectories in a special order, looping several times over the build, omitting dependencies, etc. Miller proposes another approach [16] based on one common build script which (literally) includes local scripts describing the specific dependencies of a subdirectory. This also has the advantage of addressing other drawbacks of recursive builds such as long build duration and error-prone build parallellisation.

Unless special care is taken to devise and document clear conventions, people are puzzled by the complex composition of the build scripts. The variables introduced by the configuration system aggravate things, as problems may be tied to certain configurations only. A plain "grep" no longer suffices. If a build system becomes too hard to understand and maintain, the risk of tailoring the source code structure to the build layout increases.

Another insight into the complexities of maintaining build systems comes from Robles [24], who investigated the role of various non-source code artifacts in open source software. For the KDE desktop environment (in its GBS-era), he found that there were many big atomic commits of build files, whereas one would expect commits to contain both source code and corresponding build modifications. He blames this on two things: (1) the build system needs to change very frequently; (2) build scripts are tightly coupled, so most changes percolate through many of them at once.

To summarise, support for both reverse- and re-engineering techniques is needed to maintain build systems. The next section will install a list of requirements for achieving this.

3. Requirements

To address the problems mentioned in the previous section, we list five requirements a reasonable solution should provide, as well as some important trade-offs.

3.1. Functional requirements

Visualisation — Build systems typically consist of hundreds of scripts, whether they adhere to a recursive build (section 2) or not. As a consequence, it is nearly impossible to get a complete view of the resulting system. Hence the need for a more visual representation of the *whole* system.

It is not hard to foresee that there will be a vast amount of information, so measures should be taken to make data more digestible. Filtering springs to mind (see further down), but more simple features like color coding, layouting and zooming, would also help. Interactivity is key.

Querying — It should be straightforward to query for specific information about a particular build target, to find commands, etc. Such a feature can also be used to filter information, by selecting various targets based on some user-defined criteria. Metrics can be calculated to gauge certain build characteristics.

Filtering — As mentioned before, the user should be able to filter out redundant information like unimportant files or other build parts. We need more than that, as we also want to define new views of the build, e.g. to abstract away low-level details of a build idiom, to generate a build-time view (see section 4.1 for more details) or even to recover the design of the source code. Therefore, we want to enhance filtering with more powerful abstraction capabilities.

Refactoring — Legacy build systems suffer from similar problems as legacy source code [3]: they are very hard to understand, but need to be changed continuously to cope with evolution. Refactoring of the system should therefore be possible, exploiting knowledge of the build system. In addition, simulation of the effects of refactorings would provide a valuable aid. If everything works out, the modifications can be applied to the actual build system; if not, we can just "roll back". Of course, one should take care that the refactoring remains applicable across all configurations.

Validation — Detecting bugs in the build system itself is hard. We can make a maintainer's life easier, e.g. by finding dead code or looking for circular dependencies. A lot harder is the detection of implicit dependencies (see next section) or validation of a refactoring. In general, some kind of model checking approach is needed, where style rules, idioms, error patterns, etc. can be modeled and then checked on the existing build system.

3.2. Design trade-offs

Static vs. dynamic data — An important decision is whether one wants to manipulate a model of the static build and configuration scripts, or of an actual (dynamic) build run instead. In general, it is easier to obtain data from the dynamic build (e.g. through traces) than it is to analyse the static description of the build. In addition, the actual values of variables and macros are present. Unfortunately, they correspond to just *one* run on *one* particular build platform. Instead, the static data contains all information across any supported build platform, which makes it easier to reason about and to refactor across various platforms. The complexity in reliably processing and linking build components together from the static data is, however, much higher. Common errors such as misunderstood macro expansions are particularly hard to resolve as well.

```
1 all: A.class app.jar
  A.class: A.java ; javac A.java
3 app.jar: ; jar cf $@ A.class
```

Figure 2. Example makefile.

Implicit dependencies — Consider the three rules in the makefile of Figure 2. The first one (line 1) states that target "all" requires both a file named "A.class" as well as an archive "app.jar". The former is built from a Java source file named "A.java" (line 2). The rule for the latter (line 3), however, does not explicitly specify its dependencies; it only mentions them in its command list. In this case, "A.class" is an *implicit* dependency of "app.jar". More complex situations exist where (quoted) shell commands evaluate at run-time to the actual build dependencies (another example of why it is so hard to work from the static description of build systems; see previous point). This phenomenon obscures the build, hampers incremental compilation and breaks build parallellisation. E.g. if the "app.jar"-target would be built directly, "make" will not try to remake "A.class", possibly resulting in an incorrect build.

Implicit dependencies should be tangible to the user, in order to really understand what is going on during a build.

4. Existing tools and techniques

Various categories of related work exist, but, as we will see, none of them fulfill all our requirements. Querying, refactoring and validation remain largely unexplored.

4.1. Reverse-engineering community

Qiang Tu and Michael W. Godfrey [25] have proposed the build time architectural view as a proper addition to Krüchten's "4+1" View model [15]. It is mainly targeted at documenting the high-level architecture of build systems (visualisation), e.g. to describe a new architectural style found within GCC and the Perl interpreter. In practice, the Build Time View (BTV) Toolkit accomplishes this using the grok tool [19], which filters low-level facts generated by an instrumented version of "make" (dynamic model of section 3.2). The BTV Toolkit's current prototype only

extracts build-time facts, although conceptually build time views also take source code into account. As build time views focus on conceptual reverse-engineering, *our querying, refactoring and validation requirements are not met*.

There are also quite some similarities with reverse-engineering approaches based on so-called fact extractors. In this field, data extracted from source code [18, 4, 20, 13, 5, 10, 12], object files [11, 23], etc. is stored as facts and relations between them. Using a query language, these facts can be filtered, reduced and composed (using human intervention) into a high-level architecture of a software system. Dali [13] (recently renamed to ARMIN) already exploits build-related facts in addition to source code to obtain a graph model of a system. Human experts need to derive and define patterns in this model to gradually obtain a high-level model. These patterns are expressed as SQL-queries. *None of these techniques, however, target querying, refactoring or validation*.

4.2. Re-engineering community

In [8], include dependencies of C/C++ systems are restructured in order to speed up builds. A reflexion model [21] is used to expose any divergences and absences w.r.t. a proposed source code architecture. All repair actions undertaken are used to control directory and header file restructuring. In fact, the build system is refactored by restructuring the source code using a modified GCC. As a side-effect, the build and the software architecture become easier to understand, as they are better structured now. As this technique relies on modification of the source code, however, *it can not be applied to build systems in general*.

Di Penta et al. [23] proposed a framework for renovating software based on a dependency graph of binaries and object files. Using genetic algorithms, clustering techniques and human intervention the are able to detect and throw away redundant object dependencies and clones.

4.3. Enhanced build tools

The tools in this section each target only one of our requirements. Remake is an improved GNU Make with extra tracing capabilities and a debugger. One can set breakpoints, step through the build and evaluate expressions. Another debugger named "gmd" is implemented completely using "make" macros. Tools like Antelope, AntExplorer and Openmake Build Monitor allow live visualisation of build runs. Makeppgraph creates a build dependency graph in which colors are determined by file extensions. *There is only limited filtering control, and no refactoring or validation support*. Vizant is a similar tool for Ant files. Finally, there are some tools to assist "make" users. Maketool is an IDE for makefiles providing colored logs, collapsed directories, etc. Build Audit transforms build traces in more structured HTML or text formats, while mkDoxy is a documentation tool for "make" scripts.

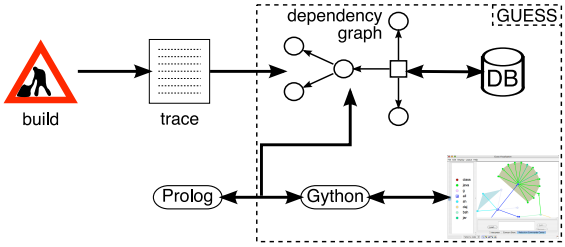


Figure 3. Outline of MAKAO's architecture.

5. Design and implementation

We will now present the implementation of a reverse- and re-engineering framework for build systems, named MAKAO (*Makefile Architecture Kernel featuring Aspect Orientation*[1]). It is designed according to the requirements set out in section 3. Figure 3 shows the architecture.

5.1. Build system representation

Our visualisation requirement of section 3.1 can easily be satisfied, as "make" is actually based on a Directed Acyclic Graph (DAG) [9] in which nodes are targets and edges represent dependencies between those targets. DAGs have lots of favorable characteristics, one of them being their natural visualisation, so we adopt them as the main model.

As for the static versus dynamic trade-off, we opt for a hybrid approach in which dynamic data is enhanced with static information such as the build rules and unevaluated targets. Problems are usually first encountered when running a build for a particular configuration, so a bottom-up approach starting from the dynamic model for that configuration makes sense. Extracting the relevant static data and linking it to the actual makefiles is therefore sufficient.

Retrieving a build's dependency graph can either be done using a modified "make" (as "BTV Toolkit" does; see section 4.1) or by parsing the trace output produced by the build tool. Because of its loose coupling, we currently use the latter option. It is important to stress that, as most build tools share "make"'s dependency-based model, most of them can be supported by MAKAO. In fact, apart from GNU Make, we already provide support for ClearMake.

To detect implicit dependencies we leverage the Bash shell's "xtrace" option, which prints every single executed command to the trace file, with all arguments expanded. It

[1] Downloadable from `http://users.ugent.be/~badams/makao/`.

then suffices to select all names with an extension (e.g. ".c") which are not listed as target or dependency of the enclosing rule. This may, of course, ignore implicit dependencies on certain files, such as Linux binaries, which typically have no extension. False positives can also occur.

5.2. Implementation

We opted to build MAKAO on top of GUESS [2], a graph exploration tool with an embedded Jython-based scripting language named Gython. Graphs can be loaded from file or from an embedded database. All nodes, edges and hulls are objects with their own user-definable attributes (name, concern, line number, etc.), which enables easy querying, navigation and refactoring of the graph.

While Gython could also be used for filtering and validation, symbiosis with a declarative rule-based approach offers more advantages [17]. We therefore integrated the SWI Prolog engine into GUESS, in which an equivalent logical representation of the graph model is kept in sync with GUESS's internal model.

5.3. Refactoring using AO

There are various ways to model build system refactorings, but we opted for an aspect-oriented [14] approach (AO). Basically, a pointcut first selects all relevant join points where advice should be woven during the build. Join points are moments during a build where one would like to enhance the existing behaviour (advice) e.g. with new commands, extra dependencies, new rules, etc. Physical weaving in MAKAO boils down to modifying the build scripts, but before doing that one can first experiment merely doing logical weaving on the in-memory build model in the tool.

For limited refactorings, MAKAO's reverse-engineering capabilities are sufficient as it is much easier to just manually change build scripts. As soon as changes need to mingle with e.g. existing command lists (tangling) or need to be distributed across various places in a context-dependent way (scattering), tool support is required. AO is a perfect fit for these kinds of refactorings.

6. Applications

In order to assess the design and implementation of MAKAO, we will now apply it on some typical build-related problems in the Kava system and the Linux 2.6.16.18 kernel. This section will report on these experiments in the order of the requirements set out in section 3.

Kava[2] is a non-profit organization of over a thousand Flemish pharmacists which has evolved into a full-fledged service provider. Some ten years ago they developed a suite of applications written in C and built using "make". Due to successive health care regulation changes, this service has been re-engineered. They also migrated from a non-ANSI C to an ANSI C platform. During this move the original manually written makefiles have been converted to an automake-based system, although some old makefiles still persist and others have been tweaked manually. This C system comprises of 272 makefiles (4683 SLOC[3]).

The Linux kernel has been used extensively by other researchers, e.g. in [5, 6]. It uses [1] a custom "make"-approach named "Kbuild" together with the "Kconfig"-framework for easy configuration of the various kernel modules. There are various frontends for the end user to select the desired components. These choices eventually end up as variables controlling the build scripts. There are about 859 build scripts ($\pm$13147 SLOC) and 345 configuration files ($\pm$51489 SLOC) involved in the 2.6.16.18 kernel.

6.1. Visualisation

Figure 4(a) shows the dependency graph (4040 nodes and 6207 edges) of Kava's build system, as presented by MAKAO. The overall layout of this graph has been taken care of by the GUESS framework. Figure 4(b) shows a more detailed view of the marked subgraph of Figure 4(a).

Every target has a color based on its concern, i.e. the type of file indicated by its extension (see the legend on top of Figure 4(b)). Edges have the same color as their destination node. MAKAO initially defines only a couple of build concerns, like .c source files and .o object files. One can add other concerns to MAKAO's list and assign a color, or even a different symbol. By doing this we can try to separate out all the different types of targets.

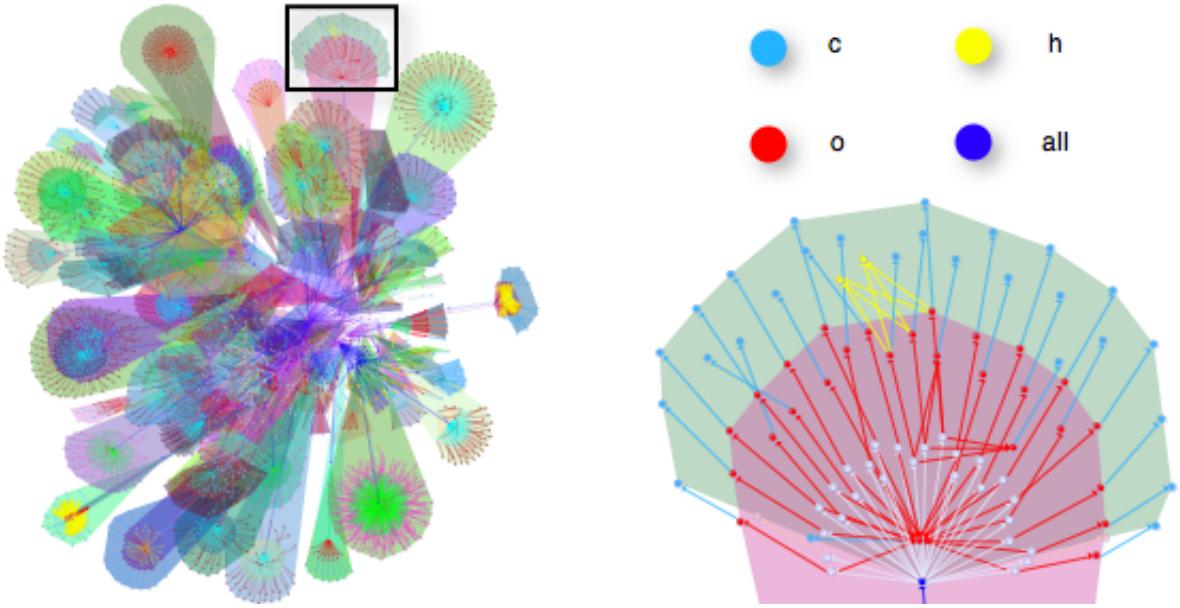
The filled polygons on Figure 4(a) and 4(b) are convex hulls, enclosing all nodes having the same value for a specific characteristic. In MAKAO, we chose as characteristic the name of the makefile specifying the target. The transparent colors of these hulls are chosen at random and convey no semantic meaning; they just aid visual recognition.

When looking at Figure 4(a), we can definitely say that it reflects a pure recursive build. Many hulls are disconnected and pointing outwards, with exactly one incoming edge and no outgoing ones. These assumptions are confirmed when browsing through the makefiles later on.

Figure 5(a) shows the dependency graph (3015 nodes and 8308 edges) of the Linux 2.6.16.18 kernel, more in particular the "bzImage"-target. This DAG looks very different from that of Kava. Apparently, there is an unknown core (light purple), depending (as expected) on object files (red) and .c files (blue) respectively. There are also two clusters of header files (yellow). These correspond to two places

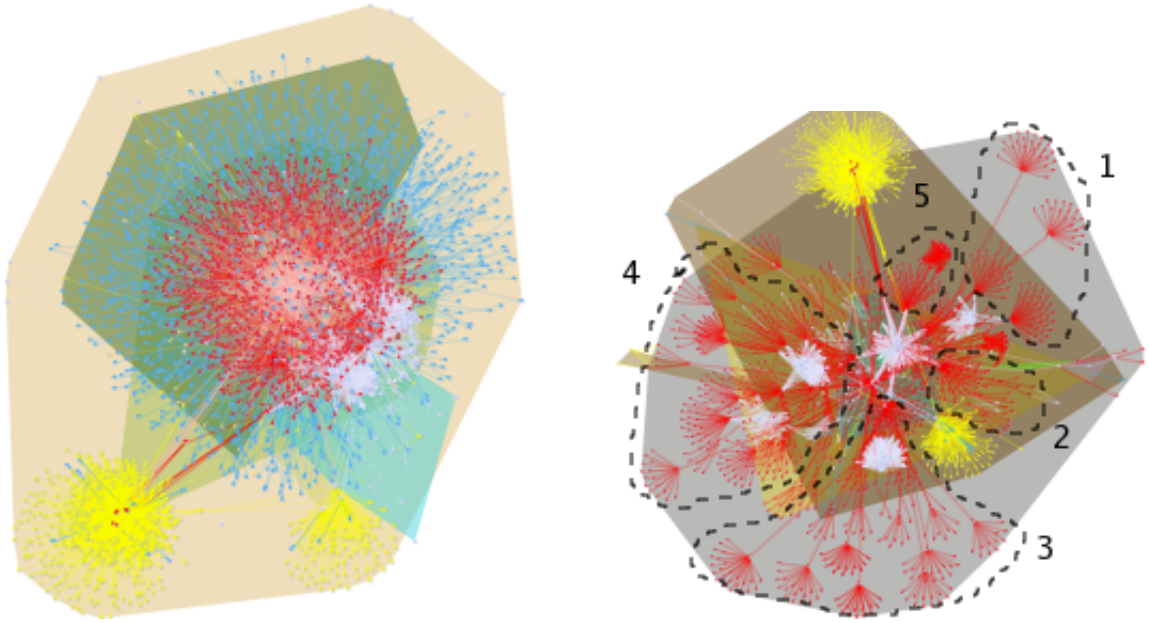
[2] The Royal Pharmacists Association of Antwerp, see `http://www.kava.be`.

[3] Calculated using `http://www.dwheeler.com/sloccount/`.



(a) (b)

Figure 4. (a) Kava's build dependency graph in MAKAO. (b) Marked subgraph in detail.



(a) (b)

Figure 5. Linux kernel 2.6.16.18 ("bzImage"), (a) before and (b) after filtering.

where custom and system header files are checked. Hulls are rather concentric and the DAG is much more dense, so this does not look like a recursive build at first sight. We will see later on (section 6.3) how we can untangle this DAG to get a better understanding.

To summarise, even without full querying support, one gets a useful qualitative view of a build system.

6.2. Querying

We now show some examples of Gython queries for finding more specific information from the graphs.

Error detection — Occasionally, we encountered some error messages during the build of the Kava system. In MAKAO, we can write the following simple Gython query to investigate this:

```
1 (error==0).visible=0
```

The expression between the parentheses gives us the so-called "node set" of all targets in which the "make" `error` status was zero, i.e. the targets which did not fail. We then hide all these targets by setting their `visible`-attribute to zero. This leaves us with a minimal subgraph starting from the main target to all failing targets. From this, the Kava developers were able to tell that the failing targets correspond to dead code. In the meantime, this has been cleaned up.

Tool mining — Knowing that the Kava system contains both .c and .ec files, we want to find out what compiler is used. By querying the `commands`-dictionary for some of these nodes' command lists, we can quickly discover the use of tools like "gcc" and "esql". To ensure that there are no nodes making use of some other compiler, we can issue the following query:

```
1 Ts=(concern=="c").inEdges.node1.findNodes()
  base=[T for T in Ts if not
3         using_tool(t,["gcc","esql"])]
```

The first expression finds all source nodes (`node1`) of edges pointing to .c targets, i.e. all targets depending on .c files. (It is clear that proper detection of implicit dependencies is important here, otherwise, we would be sure to miss some files.) Lines 2–3 then yield all targets `T` which are not using any of the compilers we already found. The `using_tool`-function internally applies Java regular expression matching on the `commands`-dictionary. Iterating this approach with newly found commands eventually gives us all source-processing tools in use.

Name clashes — A "make" process has only one global namespace shared by all targets, environment variables and macros. This can get tricky in non-recursive builds where dozens of files are included in the main makefile. MAKAO can detect these problems like this:

```
1 [c for c in groupBy(localname)
    if len(c)>1]
```

This snippet uses a list comprehension to first cluster the targets based on their local name, and then only keep the clusters with more than one target. For each returned cluster's targets we can then check the `makefile`-attribute to see if there is a real name clash.

Where do compiled objects end up? — Another frequent problem is finding out where build artifacts are stored. This is quite easy, e.g. for the file named "built.o":

```
build_dir=by_localname("built.o")[0].dir
```

In addition we might check whether multiple makefiles use this directory for their output or whether this is only one makefile's territory:

```
1 set((dir==build_dir).makefile)
```

This expression returns the set of all makefiles of which targets are produced inside `build_dir`.

6.3. Filtering

In section 6.1, we saw that the Linux DAG was rather unclear, as it was very compact (Figure 5(a)). In order to remedy this, one could try to manually collapse nodes one by one, or maybe provide a script to do this [22]. Unfortunately, this is far too coarse-grained and limited, as each system can exhibit some special idioms. The Linux build e.g. uses "shipped" targets. These are binary blobs or files generated by e.g. lex or bison. which are copied to a source file or can be overridden locally. Exploiting knowledge of these idioms leads to more effective filtering.

Filtering the build dependency graph also gradually raises the abstraction level. If no important dependencies are removed, we end up with a high-level view of the software itself as seen through the eyes of the build system. To this end, we will use a user-customisable set of logic rules to detect patterns in the graph and remove/add edges/nodes based on the results. The definitions of all these rules follow the same pattern:

```
1 rule(name, Remove, Add, MayBeLost):-
    %> rule logic.
```

Here `name` reflects the intent of the rule. `Remove` and `Add` are lists of nodes/edges which are to be removed and added, respectively. In addition, `MayBeLost` is a list of nodes which may possibly get disconnected from the main DAG, in which case they should also be removed.

Nodes and edges are represented like this:

```
  node(ID, Name, Concern).
2 edge(TargetID, DependencyID, Time).
```

We grouped the filtering rules for unravelling the Linux build into four phases, which we will now expand on.

1. Simplification — Makefiles can include other files, in which case there will be an edge from the makefile's start target to the included makefile node. We call these "meta-edges". They are useful for checking whether a build is recursive, but not for simplification. As they are at the meta-level, we consider their removal to be "semantics-preserving", i.e. mere simplification. The following rule gets rid of them:

```
  rule(eliminate_meta,[edge(A,B,Time)],[],
2       [A,B]):-
    meta(A,B,Time).
```

In this Prolog rule (named "eliminate_meta"), we remove all edges from `A` to `B` which have been flagged as being meta (line 3). No new edges or nodes are generated. We also indicate that both node `A` and `B` could end up disconnected from the DAG (line 2). The `Time` variable is used to select the correct dependency, as there may be more than one between any two given targets.

2. Reduction — While exploring the Linux DAG, one can see a node named "FORCE" on which lots of other targets depend. It turned out that this is a known GNU Make idiom to have some targets always be remade (also known as "phony" targets). It is because of this node that Figure 5(a) turns out so compact and tied together. We can easily remove this node through a Prolog rule. This is reduction rather than simplification, because the FORCE target is essential to the build's semantics. As a result, the DAG opens up somewhat and a more conventional hierarchical visualisation appears. At the same time, .c files and .o files are now separated better, giving rise to some small clusters. The hulls still look the same.

3. Abstraction — We will now take into account conventional knowledge about dependencies between object and source files. We first relate source code and object file concerns, as in:

```
1 source_object(c,o).
```

Facts for user-specific file types can easily be added. We can then write our abstraction as follows:

```
1 rule(abstraction_object,
       [edge(ObjNode,SrcNode,Time)],[],
3      [SrcNode]):-
    source_object(Source,Object),
5   node(SrcNode,_,Source),
    \+ edge(SrcNode,_,_),
7   edge(ObjNode,SrcNode,Time),
    node(ObjNode,_,Object).
```

This looks for a source node (line 5) without any dependencies (line 6) and looks up its corresponding (line 4) object file (lines 7–8), i.e. the object node depending on it. We then remove the edge between these two (lines 1–2) and name the source target as a possible orphan (line 3).

Once source files are collapsed, one can abstract up to applications by hiding object files and libraries. It is important to discriminate between applications linked directly from object files and applications depending on a source file and a bunch of object files. In the latter case, it would be safer to add an extra object dependency in between the application and source target. For brevity, we do not show the relevant logic rule.

4. Application specific idioms — At this point, the graph is sufficiently filtered to detect various idioms in it. Exploiting these, the graph can be abstracted further. We easily detect a particular Linux build-idiom named "composite objects" [1]. The modules making up the kernel image correspond to giant object files linked from various smaller object files. It is easy to detect these composite objects, as they exclusively depend on (lots of) other object files, and their parent node is always the same one (named `__build`). The upper four nodes of Figure 6 illustrate this.

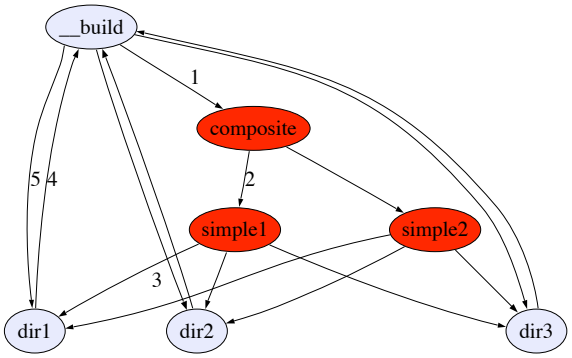


Figure 6. Circular dependency chain.

In combination with these composites, a phenomenon we named "circular dependency chain" occurs. Figure 6 shows the basic pattern and its control flow. The central (phony) `__build` target depends on all composite objects (arrow 1), which of course depend on a number of simple objects (arrow 2). The strange thing is that the latter depend on every possible subdirectory of their enclosing directory (arrow 3), and each of them points back to `__build` (arrow 4). To top it off, `__build` depends again on each subdirectory (arrow 5).

This circular dependency chain is actually a clever iteration strategy the Linux developers added to their recursive build in order to get rid of the side-effects explained in section 2. Indeed, to avoid that the evaluation order of targets is of influence (cf. Figure 2), every subdirectory target will rebuild `__build` to let earlier targets notice intermediate changes. "Make"'s time stamp mechanism makes sure that this iterative algorithm stops.

With this idiom in mind, breaking up all edges labeled "4" and "5" will open up the graph completely, and result in the much more structured Figure 5(b). We can identify various subsystems: (1) network support, (2) kernel, (3) file system code, (4) drivers and (5) architecture-dependent things. This can be the starting point for further filtering.

6.4. Refactoring

Having recovered all necessary knowledge either by visualisation, querying or filtering, one can now refactor the build system using AO techniques (see section 5.3). As an example, we integrate a tool called Aspicere [26] into Kava's build system based on the information gathered in sections 6.1 and 6.2. Aspicere should be run before each C

source file's compilation. We will now illustrate how to do this for basic .c files.

First, we search the targets `T` manipulating .c files and look up the relevant commands `C` in `T`'s command list:

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

For each triplet, we then compose a before-advice, i.e. commands which should be invoked before every `C`:

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

We simply want to feed Aspicere with the preprocessed .c file. Preprocessing can be done by adding a preprocessing flag ("-E") to command `C` and redirecting output (line 7). Line 8 contains the actual invocation of Aspicere.

Finally, the MAKAO weaver should weave the advice in front of each command captured in `base`, both in-memory as well as in the proper build scripts:

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

After weaving (lines 11–14), the `commands`-dictionary reflects the refactored command lists. If the advice had introduced new targets, these would show up too. This is the so-called "logical weaving", as it emulates a build refactoring without touching the real build scripts. One can still undo changes, modify the pointcut and/or advice and reweave.

If the refactoring behaves as expected, one can execute a Perl script generated during (logical) weaving. This will modify the actual build scripts, where needed, to complete the physical weaving. The script also has an undo option.

6.5. Validation

We want to check that the refactorings from the previous section did not introduce any errors. Some validation can be done using Gython scripts, but this is often slow (quadratic complexity or worse). That is why we prefer to make use of the Prolog bridge to describe and check certain invariants which represent common mistakes or style abuses.

As a simple example, finding unused targets boils down to the following Prolog rule, which finds all nodes from which no edge starts and in which no edge arrives:

```
unused(Target):-
  node(Target,_,_),
  \+edge(Target,_,_), \+edge(_,Target,_).
```

A more advanced kind of validation could look at application nodes which depend both on a source and its corresponding object node without any connections between the two. Such an error is possibly introduced by implicit dependencies. Other validation rules may focus on redundant dependencies between two targets (also transitively), detect loops or non-object nodes pointing to source files, etc.

7. Future work

We aim to use MAKAO to investigate whether build systems mimic the enclosing software's architecture and, if so, how well. Another question is whether manually written build systems perform better on this than generated ones.

Currently, we only base ourselves on build scripts. Our goal is to be able to access configuration data too, e.g. autoconf's config.log-file. Then, the weaver is able to change the actual templates or configuration files instead of the generated makefiles. This could result in some kind of configuration simulator, in which various configurations can be tested on virtual files. At the same time, a static build script parser is another way to accommodate variant checking.

As a lot of reverse-engineering tools apply Holt's Tuple-Attribute notation [19], we want to investigate this format in order to add facts from source code extractors to MAKAO. This could lead to another solution for the implicit dependencies problem, and to better build validation.

We would also like to check out how to couple BTV Toolkit's modified GNU Make to MAKAO, e.g. to see whether this would work better in the realm of parallel builds, as the build trace is not as reliable in such cases.

Finally, we will continue work on validation rules in order to more thoroughly assess refactorings.

8. Conclusion

Maintenance of build systems has proven to be hard due to specific build tool implementation choices and scalability problems. Based on this, we have specified five functional requirements and a couple of design trade-offs for tool support. None of the existing work tackled all of the requirements, especially querying, refactoring and validation, so we designed and implemented our own: MAKAO.

At its core, MAKAO offers a flexible Directed Acyclic Graph model of a dynamic build, which can be queried and filtered imperatively (Gython) or declaratively (Prolog). To address refactoring, we proposed an aspect-oriented approach combined with logic rule-based validation. For each requirement, we illustrated MAKAO's use on typical build problems in the Kava system and Linux 2.6.16.18 kernel.

MAKAO showed itself as a useful, practical tool for re(verse)-engineering a build and for further build system

research. The work on validation shows great promise for quality assurance of build systems.

9. Acknowledgements

The authors want to thank Bernard De Ruyck and the other people from Kava for their generous support. We would also like to thank Andy Zaidman (TU Delft) for his valuable input. Bram Adams is supported by a BOF grant from Ghent University. Kris De Schutter received support within the Belgian research project AspectLab, sponsored by the IWT, Flanders.

References

[1] *Linux kernel build documentation*, linux 2.6.16.18 edition.

[2] E. Adar. GUESS: a language and interface for graph exploration. In R. E. Grinter, T. Rodden, P. M. Aoki, E. Cutrell, R. Jeffries, and G. M. Olson, editors, *CHI '06: Proceedings of the 2006 Conference on Human Factors in Computing Systems*, pages 791–800. ACM, 2006.

[3] K. Bennett. Legacy systems: Coping with success. *IEEE Software*, 12(1):19–23, 1995.

[4] T. J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, 1989.

[5] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: its extracted software architecture. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 555–563, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[6] J. Champaign, A. Malton, and X. Dong. Stability and volatility in the linux kernel. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 95, Washington, DC, USA, 2003. IEEE Computer Society.

[7] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, 1990.

[8] H. D. Fard, Y. Yu, J. Mylopoulos, and P. Andritsos. Improving the build architecture of legacy C/C++ software systems. In *FASE '05: Proceedings of Fundamental Approaches in Software Engineering*, pages 96–110, 2005.

[9] S. I. Feldman. Make-a program for maintaining computer programs. *Softw., Pract. Exper.*, 9(4):255–65, 1979.

[10] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Syst. J.*, 36(4):564–593, 1997.

[11] A. E. Hassan, Z. M. Jiang, and R. C. Holt. Source versus object code extraction for recovering software architecture. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 67–76, Washington, DC, USA, 2005. IEEE Computer Society.

[12] D. Hou and H. J. Hoover. Source-level linkage: Adding semantic information to C++ fact-bases. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 448–457, Washington, DC, USA, 2006. IEEE Computer Society.

[13] R. Kazman and S. J. Carrière. Playing detective: Reconstructing software architecture from available evidence. *Autom. Softw. Eng.*, 6(2):107–138, 1999.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97: Proceedings of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

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

[16] P. Miller. Recursive make considered harmful. *Australian UNIX and Open Systems User Group Newsletter*, 19(1):14–25, 1997.

[17] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong. Reverse engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 47–60, New York, NY, USA, 2000. ACM Press.

[18] H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 80–86, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[19] H. A. Müller, S. R. Tilley, and K. Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *CASCON '93: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research*, pages 217–226. IBM Press, 1993.

[20] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. Methodol.*, 5(3):262–292, 1996.

[21] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28, New York, NY, USA, 1995. ACM Press.

[22] S. Neginhal and S. Kothari. Event views and graph reductions for understanding system level C code. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 279–288, Washington, DC, USA, 2006. IEEE Computer Society.

[23] M. D. Penta, M. Neteler, G. Antoniol, and E. Merlo. A language-independent software renovation framework. *J. Syst. Softw.*, 77(3):225–240, 2005.

[24] G. Robles. *Software Engineering Research on Libre Software: Data Sources, Methodologies and Results*. PhD thesis, Universidad Rey Juan Carlos, February 2006.

[25] Q. Tu and M. W. Godfrey. The build-time software architecture view. In *ICSM '01: Proceedings of the 17th International Conference on Software Maintenance*, pages 398–407, 2001.

[26] A. Zaidman, S. Demeyer, B. Adams, K. De Schutter, G. Hoffman, and B. De Ruyck. Regaining lost knowledge through dynamic analysis and aspect orientation. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 91–102, Washington, DC, USA, 2006. IEEE Computer Society.